Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

INF/01

Ph.D. Thesis

Measuring the Semantic Integrity of a Process Self

Daniele Sgandurra

Supervisor
Fabrizio Baiardi

May 21, 2010

Every act of creation is first of all an act of destruction.
Pablo Picasso

Abstract

The focus of the thesis is the definition of a framework to protect a process from attacks against the process self, i.e. attacks that alter the expected behavior of the process, by integrating static analysis and run-time monitoring. The static analysis of the program returns a description of the process self that consists of a context-free grammar, which defines the legal system call traces, and a set of invariants on process variables that hold when a system call is issued. Run-time monitoring assures the semantic integrity of the process by checking that its behavior is coherent with the process self returned by the static analysis. The proposed framework can also cover kernel integrity to protect the process from attacks from the kernel-level.

The implementation of the run-time monitoring is based upon introspection, a technique that analyzes the state of a computer to rebuild and check the consistency of kernel or user-level data structures. The ability of observing the run-time values of variables reduces the complexity of the static analysis and increases the amount of information that can be extracted on the run-time behavior of the process. To achieve transparency of the controls for the process while avoiding the introduction of special purpose hardware units that access the memory, the architecture of the run-time monitoring adopts virtualization technology and introduces two virtual machines, the monitored and the introspection virtual machines. This approach increases the overall robustness because a distinct virtual machine, the introspection virtual machine, applies introspection in a transparent way both to verify the kernel integrity and to retrieve the status of the process to check the process self.

After presenting the framework and its implementation, the thesis discusses some of its applications to increase the security of a computer network. The first application of the proposed framework is the remote attestation of the semantic integrity of a process. Then, the thesis describes a set of extensions to the framework to protect a process from physical attacks by running an obfuscated version of the process code. Finally, the thesis generalizes the framework to support the efficient sharing of an information infrastructure among users and applications with distinct security and reliability requirements by introducing highly parallel overlays.

Acknowledgments

First of all, I would like to express my sincere gratitude to my supervisor, Prof. Fabrizio Baiardi, who has played a major role in my growth as a researcher with his continuous support and professional supervision throughout my activities.

I am very grateful to Thomas Jensen and Sebastian Mödersheim, who reviewed this thesis, for their insightful comments and suggestions that have been invaluable to improving my thesis. Special thanks go to my internal referees, Nicoletta De Francesco and Giorgio Levi, who provided many helpful comments.

Several people deserve special mention for their contributions: *in primis*, Dario Maggiari, for his immense help during the design and implementation of both the static and the run-time components of the framework; Francesco Tamberi, for contributing with several fruitful ideas to the development of the run-time architecture; Gaspare Sala, for his contributions to the design and implementation of the virtual community framework; Diego Cilea, for having contributed to the design and implementation of the remote attestation framework; Fabio Campisi, for supporting me during the development of the introspection functions; Diego Zamboni for his patience, invaluable help and support during the design and implementation of the context-agent framework. Finally, I also wish to thank Alberto Daniel, Stefano Paganucci and Federico Tonelli for their precious contributions.

I would like to thank all the PhD students that I have met at the Department for making my life so enjoyable: Michele Albano, Davide Cangelosi, Giulio Caravagna, Cristian Dittamo, Gabriele Mencagli, Luca Nicotra, Igor Nitto among the others.

Last, but not least, I would like to thank my parents for always supporting me.

Contents

List of Figures

List of Tables

Introduction

Before describing the goals of the thesis and the methodologies that it proposes, we briefly review some terminologies and some classes of attacks against a computer system. A software *bug* is an error in a program, due to either a programming error or an erroneous specification, that prevents it from behaving as intended. Some bugs may cause the program to crash, freeze or halt and may be undetected for a long time. A *vulnerability* is a bug that has security implications for the overall system, i.e. it may enable an attacker to violate the security policy. An *exploit* is a fragment of code and data that takes advantage of a vulnerability to violate the security policy of a system. Finally, an *attack* executes an exploit to effectively take advantage of a vulnerability. By composing several simple attacks, an attacker implements a complex attack to achieve a goal of interest, which can be full system control or denial of service.

At a high level, we can categorize attacks against a computer system into two wide classes of attacks against the integrity of distinct components:

attacks against user-level processes, which allow an attacker to insert some code into a process and, eventually, to diverge the original control-flow to execute the injected code;

attacks against the kernel, which modify some kernel functionalities. Usually, they are the final step of a complex attack because they require that the attacker has already gained root privileges. If successful, they modify the behavior of the kernel to hide any sign of the previous steps of the complex attack against the system.

In the following, we describe buffer overflows and rootkits, the most popular attacks that belong, respectively, to the first and second class.

Buffer Overflow

A buffer overflow [61] attack exploits a software vulnerability due to a programming error so that a process may store data beyond the boundaries of a fixed-length

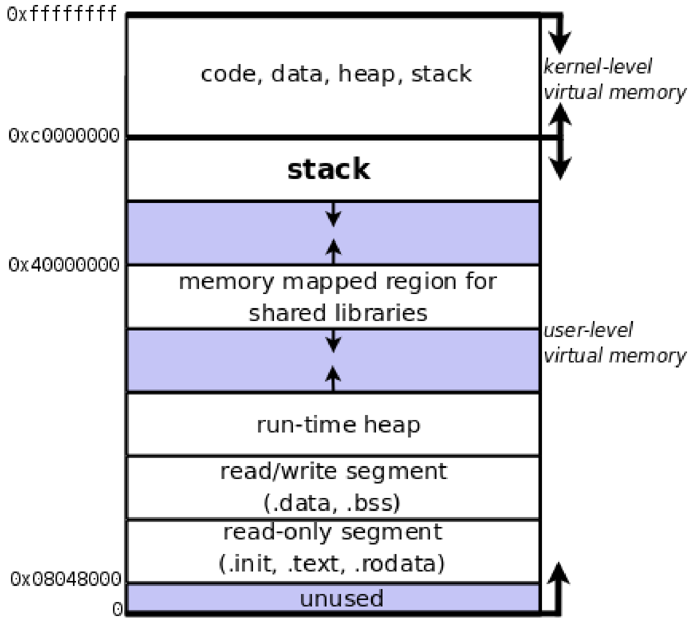

0xffffffff
code, data, heap, stack
kernel-level virtual memory
0xc0000000
stack
0x40000000
memory mapped region for shared libraries
user-level virtual memory
run-time heap
read/write segment (.data, .bss)
read-only segment (.init, .text, .rodata)
0x08048000
unused
0

Figure 1: Process Virtual Memory Layout

buffer. This results in the overwriting of adjacent memory locations, which may store variables or control-sensitive data structures. Eventually, this erroneous memory update may crash the process or cause the execution of malicious code. Usually, to implement a buffer overflow, an attacker sends a large amount of data to a network process, coded in a type unsafe language that lacks a native array-bound checking mechanism. If the listening process stores the received data in an undersized stack buffer, it may overwrite data on the call stack, such as the function's return pointer. If the data to be stored in the buffer has been properly crafted, the attacker may overwrite the value of the return pointer with a value of her choice and transfer the control to malicious code in the data itself. This part of the attack is a *control-hijacking attack* [221], where the attacker diverges the program control-flow to execute instructions in the transmitted data.

From a historical point of view, stack-based buffer overflows were already discovered and analyzed in 1972 [7]. In 1996, [4] described in full detail how to exploit stack-based buffer overflow vulnerabilities. The Morris worm (1988) is the earliest known exploitation of a buffer overflow that spread over the Internet by exploiting a flawed version of `fingerd` [224]. More recently, at least two worms have exploited buffer overflows to compromise a fairly large number of systems over the Internet: in 2001, the "Code Red Worm" exploited a buffer overflow in Microsoft's Internet Information Services, while in 2003 the "SQL Slammer Worm" compromised a large number of hosts running Microsoft SQL Server 2000.

The ability of implementing a buffer overflow attack is strictly related to the process virtual memory layout. As shown in Fig. 1, on x86 architectures running

Linux, the private stack of a user process grows from high memory addresses to lower ones. The C function calling conventions are relevant as well, since they rule:

(i) the order to allocate the function parameters;

(ii) where parameters are placed, either on the stack or in the processor's registers;

(iii) whether the code of the caller or the one of the callee is responsible for unwinding the stack on return.

A buffer overflow vulnerability is the joint effect of the process virtual memory layout, in particular the stack growing from higher memory addresses to lower ones, and the function calling conventions that impose to store local variables at memory addresses that immediately precede the return address. For instance, several string functions in C do not perform bound checking and can easily overwrite the bounds of the buffers they operate upon. Most of these functions are legacy C library functions, such as the infamous `strcpy()` and `fgets()`.

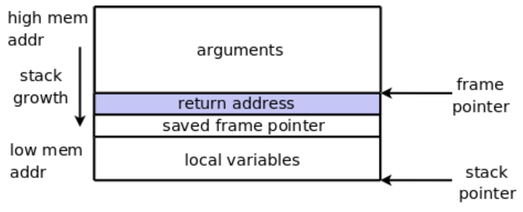
As an example, in the following C fragment

```c
void foo(char *str)
{
   char buff[5];
   strcpy(buff, str);
}
int main(int argc, char **argv)
{
  foo(argv[1]);
}
```

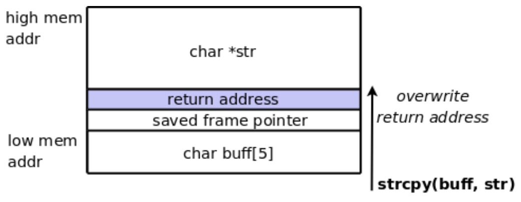
the `foo()` function copies the string pointed by the argument `str` into the local variable `buff` that can store at most five characters (actually, four characters plus the trailing `nul` character). If the string is longer than five characters, eventually, during the copy, some characters will overflow the receiving buffer. Figure 2(a) shows the C calling conventions that dictate that local variables are stored at lower memory addresses than the return address. For these reasons, in the previous example, the `strcpy()` may end up overwriting the return address (see Fig. 2(b)). Furthermore, since `argv[1]` is the first command line argument transmitted to `main`, and it is directly passed as a parameter to the vulnerable function, the attacker can craft the parameter to overflow the buffer and overwrite the return address. Then, if the parameter codifies some malicious code, the execution of the original program resumes at the return address specified by the attacker and it results in a malicious behavior of the process. Usually, according to the privileges of the attacked process, the malicious code spawns a root shell, which gives the attacker full control of the system through privilege escalation.

Some known countermeasures to buffer overflow attacks are:

- avoid the use of vulnerable functions, such as `strcpy()`, `fgets()` and the likes, that operate on `nul`-terminated strings and that perform no bounds checking;



(a)



(b)

Figure 2: Function Call Conventions (a); Overwriting the Return Address (b)

- prevent the execution of code on the stack. Since, usually, the malicious code is stored on the stack, one simple countermeasure invalidates the stack to execute any instruction, so that any attempt to execute code in the stack results in a segmentation violation. However, this countermeasure is not fully general because some compilers, such as GCC, use trampoline functions that require an executable stack and, furthermore, it does not prevent heap-based buffer overflows;
- apply static checks, as an example, by enabling compilers to produce warnings on the use of unsafe invocations;
- enforce run-time checks: as an example, *StackGuard* [62] prevents *smash stacking attacks* by protecting the return address on the stack by placing a *canary* word (usually, a random value) next to this address and by generating code to check this value before the function returns. If the canary has been altered, then the stack has been successfully overflowed. In another scenario, the access of a process to the run-time support is restricted. An example is *libsafe* [237], a library that secures calls to unsafe functions. Further approaches apply a *sandbox*, i.e. a mechanism that provides a tightly-controlled set of resources to the running process [3].

These countermeasures may be integrated with an Intrusion Detection Systems (IDS) [64] that provides an on-line auditing capability to detect attacks or anomalous events. Most IDSes monitors the execution of a program or of a system and raises an alarm if it diverges from a statistical model that describes the expected behavior of the monitored entity. For example, a host-based anomaly IDS closely monitors the activities of an application process and, if any of those activities deviates from the training-based statistical model, either it terminates the process or flags its activities as suspicious. A further common way to model the acceptable behavior of an application is to monitor the sequence of system calls invoked by the corresponding process [228, 116]. If we assume that an attacker can inflict damage to the system only by invoking some system calls, then by monitoring the system calls that a process issues, an IDS can detect and prevent any malicious activities. This approach poses the problem of deducing a statistical model that faithfully represents the behavior of the application and that also minimizes both false positives and negatives. In fact, one of the most complex issue to be faced is the false alarm rate, which dramatically limits the usefulness of a statistical model [11]. It is worth stressing that, due to the large number of system calls that a process invokes, even a fairly low false alarm rate may result in a very large number of false alarms and in a useless IDS that “cries wolf”.

Rootkits

A distinct class of attacks includes those attacks that modify some functionalities of the operating system (OS) kernel. In this way, *an attacker can modify the expected behavior of a program by altering the functions implemented by the underlying kernel rather than the program itself.* Usually, an attacker has to gain root privileges to modify the kernel behavior. A *rootkit* is a collection of software tools that modify the kernel behavior so that: (i) system calls return bogus information about the status of the system to hide the traces of the compromise; (ii) there is a hidden path that allows the attacker to easily access the system at anytime. Since any security strategy aimed at obtaining full control of a computer system, for attacking or defending it, has to be applied at the lowest possible system level, rootkits have gradually evolved from user-level rootkits to kernel-level ones [225]. In fact, rootkits of the first generation run at the user-level and, usually, modify only critical system binaries to hide specific processes or files owned by the attacker. Hence, they can be easily detected by comparing the hash value of the original file against the compromised one [135]. Rootkits of the second generation enable the attacker to insert code into the kernel, e.g. to modify the behavior of a system call.

Static Code Analysis

There are several alternative approaches based upon static analysis to prevent vulnerabilities or detect attacks. Some tools implement formal analysis to prove properties of interest of a program, for example that its behavior matches that of its specification [115]. Other tools may help the developer to avoid likely coding errors or may locate potentially vulnerable code [77].

The approach that we propose considers a static analysis that exploits the concepts of control-flow analysis and invariants to detect attacks. A basic block is a linear sequence of program instructions with just one entry point, i.e. the first instruction executed, and one exit point, i.e. the last instruction executed. A control-flow analysis represents a program as a control-flow graph, i.e. a directed graph where the nodes represent basic blocks and the edges control-flow transfers. Another static analysis that the thesis exploits generates invariants, i.e. predicates on the program variables. Invariants play a central role in static analysis and run-time monitoring [74], because they can be exploited to verify some software properties, such as type safety, or to check at run-time a set of constraints. In the approach proposed in this thesis, integrity checks on a component are implemented through invariants that are automatically inferred through data-flow analysis or abstract interpretation. If the component is a user process, an invariant is defined on the process state and it involves the values of both program variables and the program counter. Alternative approaches deduce invariants either by monitoring the program execution [75] or by applying data-mining techniques to the source code [5].

Data-flow analysis defines a set of techniques that derive information about the flow of data along program execution paths. Each analysis couples every program point with a value that represents an abstraction of the program states that flow across that point. Alternative definitions of this state are possible. For example, in the case of reaching definitions, the value of a variable is represented by the subset of the program statements that produce a value for that variable. To generate invariants, an analysis should couple each point in the program with the exact set of definitions that can reach that point.

Abstract interpretation [58, 59, 60] is another framework to formally define an analysis that returns invariants. An abstract interpretation over-approximates the behavior of the program by modeling the effects of every statement on an abstract machine. The operations of this machine abstract the various language constructs. In general, an abstract interpretation associates each program point with a mapping between each variable and an abstract value, i.e. a value in the abstract domain. This value constrains the concrete values that the variable can assume and, as a consequence, it defines an invariant on the concrete values at that point. As an example, if the abstract domain is defined in terms of ranges of concrete values, each abstract value may be transformed into an invariant that expresses that the actual value belongs to the concretization of the abstract range.

Overview of the Proposed Framework

The goal of this thesis is to define and protect the expected run-time behavior of a process, i.e. the *process self*, from those attacks that modify these expected properties of the process [243, 245, 149, 148]. To define mechanisms to detect these attacks, the thesis proposes a methodology that integrates a static analysis and run-time checks. The static analysis of the program approximates the process self by returning a description of the behavior of a process in terms of valid traces of system calls that the process may issue and invariants on program variables that hold when a system call is invoked. Run-time components compare this description against the actual behavior and signal any difference. The proposed approach considers attacks that result in a process behavior that is inconsistent with the process self. This behavior is usually due to the insertion of malicious code into the running process. Since this code is not present in the original program, its execution results in process behavior different with respect to the one codified by the source code.

The static analysis can be formally described as the computation of a context-free grammar that describes the legal sequences of system calls and of invariants that hold when a system call is issued. Its correctness may be proved by abstract interpretation techniques. Then, at run-time, the system calls that a process invokes are traced to verify that:

1. the values of the process variables satisfy the invariant coupled with the system call;

2. the system call trace is a prefix of at least one string of the context-free grammar.

To apply these checks in a robust and unobtrusive manner, we exploit *virtualization technology*. This technology does not require the introduction of specialized hardware units to apply the checks and it increases the robustness of the overall approach by executing two virtual machines concurrently, namely the monitored virtual machine and the introspection virtual machine. The first virtual machine executes the system that runs the monitored process, while the second one is a privileged machine that inspects the state of the monitored machine through virtual machine introspection, which is a technique that directly accesses the status of the components allocated to the monitored machine. By accessing the memory of the monitored virtual machine, the monitoring virtual machine can evaluate invariants on the state of a process on this machine to assure that the process self is consistent and the underlying kernel has not been attacked. This last check is required to protect the integrity of the kernel against malicious modifications, so that we can guarantee that system calls behave in a consistent manner. The protection of the kernel is fundamental because, by subverting the kernel, an attacker can alter the behavior of a process even if its system call trace and the invariants are coherent with the output of the static analysis.

Even if the original reason in favor of virtualization is that introspection can be implemented in a transparent way, without modifying either the process or the underlying OS, this thesis also shows that the joint adoption of the proposed framework and of virtualization offers further advantages to develop a highly robust information and communication technology (ICT) system. In particular, a key advantage of virtualization is the ability of strongly reducing the cost of increasing the number of system nodes. This, in turn, can be exploited to minimize the amount of sharing among nodes and applications by defining highly parallel virtual networks that increase the overall system robustness.

To summarize, the focus of the thesis is on the definition of the expected run-time properties of a process and on the definitions of transparent, i.e. non obtrusive, checks on these properties, by integrating a static analysis and run-time monitoring based on virtual machine introspection. The static analysis builds a grammar that defines the process self as a a language where the terminal symbols are pairs that consist of a system call and an invariant that must hold when the system call is issued. Each string of the language describes the system call sequence and the variable values of distinct execution of the program. Run-time tools detect attacks against the process self by checking that the actual sequence of system calls corresponds to one string of the language and that each invariant holds when the corresponding call is issued. The adoption of virtualization supports not only a fully transparent implementation of the proposed framework, but also the definition of other strategies to increase the overall system robustness.

Main Contributions of the Thesis

The main contributions of the thesis include:

- the definition of the process self in terms of a context-free grammar of system calls and invariants on the process state; this solution merges the ability of constraining the sequence of system calls with that of coupling memory assertions with such calls and results in a high detection capability;
- the definition of PsycoTrace, a robust framework to check the process self in a fully transparent way. PsycoTrace does not require modification either to the monitored process or the underlying OS so that the process is unaware of being monitored. PsycoTrace can also protect kernel integrity with a high degree of robustness;
- the definition of a mechanism to bridge the semantic gap by transparently injecting an agent into the memory of a virtual machine;
- the definition of a framework that extends PsycoTrace to remotely attest the integrity of a node willing to join an overlay that generalizes the Trusted Platform Module (TPM) by applying granular checks on the integrity of a node that also consider the behavior of the node;
- the definition of the notion of system block and its adoption to both increase the accuracy and reduce the complexity of a static analysis to compute invariants of a program;
- an extension of PsycoTrace to protect a process against physical attacks by a novel code-obfuscation strategy that exploits virtualization to effectively split the program logic between two virtual machines. The first machine stores the system blocks of the original program, whereas the second virtual machine stores the system block graph and the keys to decrypt the blocks. Moreover, the second virtual machine applies introspection to continuously encrypt and decrypt memory regions in the other virtual machine according to the system block graph;

- the definition of a strategy to manage and protect an ICT infrastructure shared among users and applications with distinct trust and reliability levels. The strategy exploits highly parallel overlays of virtual machines, where each virtual machine is an instance of a specialized template customized to run a small set of software components.

Outline of the Thesis

The thesis is structured as follows:

Part I: Background

Chapter 1: Measuring the Semantic Integrity
This Chapter deepens and formalizes the notion of program self, semantic integrity and the models underlying static analysis by describing the extraction of a model that characterizes the normal behavior of the program. Moreover, it introduces PsycoTrace, the framework that we have defined.

Chapter 2: Virtualization-based Security
This Chapter discusses virtualization technology, with emphasis on virtual machine introspection, and some applications of virtualization in the field of security.

Chapter 3: Related Works
This Chapter discusses the related works with respect to sense of self, virtualization for security, TPM and remote attestation, code obfuscation and collaborative virtual environments.

Part II: Principles and Implementation

Chapter 4: Description of the Process Self
This Chapter discusses the description of the process self in terms of a context-free grammar, which defines the legal system call traces that the process may execute, and invariants that hold when a system call is issued. The static tools that build the description are presented as well. The references for this Chapter are [16, 15].

Chapter 5: Run-Time Architecture
This Chapter describes PsycoTrace run-time architecture. This architecture exploits virtual machine introspection first of all to check that the current trace is coherent with the grammar and to evaluate invariants at each system call invocation. Virtual machine introspection is also applied to monitor the kernel to detect modifications by an attacker trying

to insert and execute arbitrary instructions to alter the behavior of the kernel. Moreover, the Chapter presents a methodology and an implementation of mechanisms to transparently inject, and protect, an agent into a running virtual machine through virtual machine introspection. The references for this Chapter are [18, 19] as far as concerns kernel integrity and [233, 13] for the process self integrity.

Part III: Applications of the Proposed Approach

Chapter 6: Remote Attestation of Semantic Integrity
This Chapter presents a framework (VIMS) to protect and remotely attest the integrity of a system by integrating an initial attestation and a continuous monitoring to discover malware. VIMS is a framework based upon PsycoTrace that considers not only the configuration of the system to be attested but also its semantic integrity. The reference for this Chapter is [12].

Chapter 7: Code Obfuscation in a Virtual Environment
This Chapter discusses a new approach for code obfuscation that stems from virtualization and PsycoTrace run-time architecture that can be applied to protect a virtual machine from physical attacks aiming to access its program or its data. This solution maps the control-flow graph of the program into a system block graph, where a block is any portion of program in-between two consecutive system calls. The reference for this Chapter is [63].

Chapter 8: Trusted Overlays of Virtual Communities
This Chapter presents Vinci, an architectural framework that exploits virtualization and PsycoTrace to share in a secure way an ICT infrastructure among a set of users with distinct trust levels and reliability requirements. The references for this Chapter are [205, 17, 21, 20, 14].

Part I

Background

Chapter 1

Measuring the Semantic Integrity

This chapter introduces *PsycoTrace*, a framework aimed at the definition of a robust system to defend the integrity of a process by integrating static and run-time tools. Static tools analyze the source code to define the process self [228], i.e. the essential characteristics of the process that describe its correct behavior. Run-time tools monitor a process execution through introspection by accessing the values of the process variables and other information in the process status.

1.1 Process Self

A key issue of this thesis is the definition of the properties of a process that can be extracted from the program that the process is currently executing. In general, a process is an OS concept that involves dynamic properties, such as signals, whereas a program is a static notion related to a programming language. A strong correlation between a process and a program is established as soon as the process overwrites its address space with the code of the program that it executes. By doing so, the process also inherits some of the program's properties.

Definition (Process Self)**.** *The properties of a process that determine its run-time behavior define the* process self*.*

We assume that an attack against a process results in a modification to the process self, i.e. if the process current behavior deviates from the process self then the process code has been altered by an attack.

Definition (Measuring the Semantic Integrity)**.** *The act of defining the process self and of monitoring the actual process behavior to assure that it is coherent with the process self is referred to as* measuring the semantic integrity *(see Fig. 1.1).*

Since we are only interested in defining and checking the properties that can be extracted from the program, the process self does not include some OS-based properties, such as signals, scheduling, priority. Hence, when monitoring the process current behavior at run-time, these OS-based properties have to be excluded.

Definition (P). *P is a generic process that we want to protect.*

Definition ($Self(P)$). *$Self(P)$ refers to the process self of P.*

Definition ($SourceCode(P)$). *$SourceCode(P)$ refers to the source code of the program executed by P.*

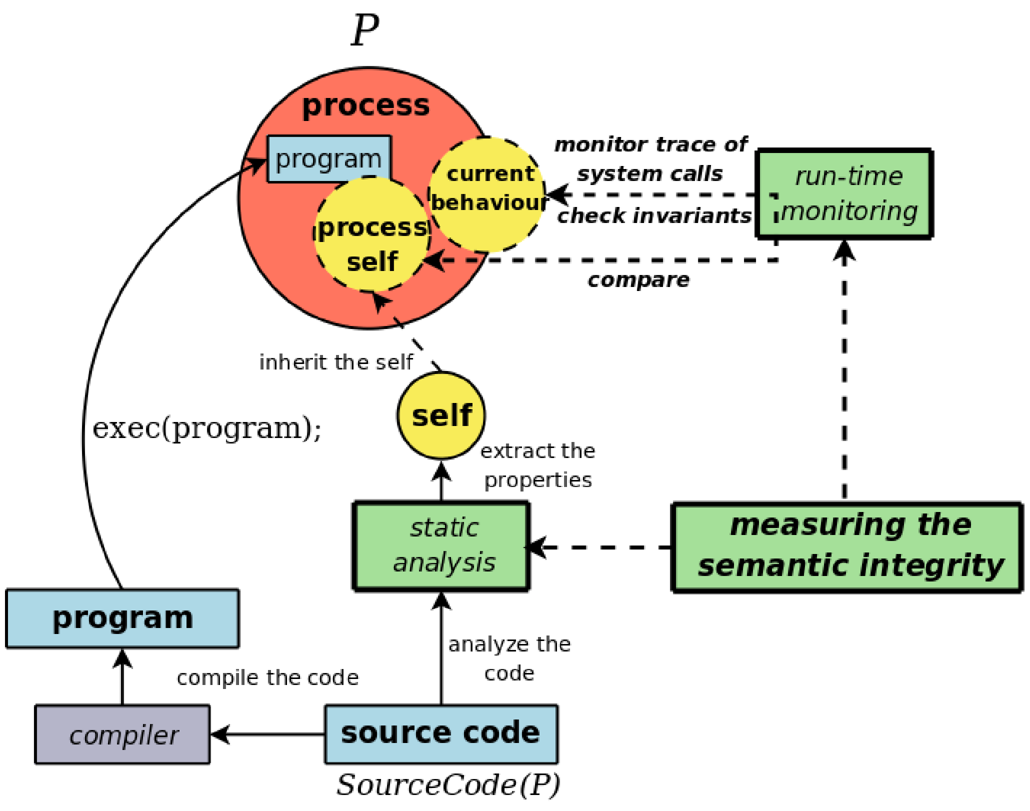


Figure 1.1: Process Self

1.2 Description of the Process Self

Alternative definitions of a process self are possible, each offering a distinct ability of detecting inconsistencies, and hence attacks, between the process self and the actual process behavior. Each definition corresponds to a distinct monitoring overhead. To offer different trade-offs between detection capability and overhead, PsycoTrace supports several alternative definitions of the process self. As previously discussed, system calls are the common base of all the strategies. Currently, the following descriptions are supported:

1. hashing or memory invariants; it defines memory invariants to be evaluated anytime P issues a given system call;
2. forbidden calls: it defines the set of system calls that P cannot issue;
3. forbidden parameters: it defines the set of system calls that P cannot issue or assertions on parameters it cannot transmit to a call;
4. allowed calls: it defines the set of system calls that P can issue and pairs each call with assertions on its parameters;
5. enriched traces: an enriched trace describes the sequence of system calls that P issues in one execution; each call may be coupled with a memory assertion. A set of enriched traces fully describes alternative legal behaviors of P.

Any strategy applies distinct measurements and results in a distinct attack detection capability. Strategies 1, 4 and 5 implement a default-deny approach that defines legal system calls, whereas 2 and 3 implement a default-allow strategy that describes forbidden calls. In general, default-allow strategies are more permissive, e.g. have a lower detection capability, than default-deny strategies. As a counterpart, default-allow strategies simplify the definition of P self by enabling the security policy to directly define the calls to be forbidden. Complexity increases for default-deny strategies that apply static tools to compute the expected behavior. These tools can be applied only if $SourceCode(P)$ is available and they return a description of traces of P that strictly includes those that P actually produces, so that no false positive arises but false negatives cannot be avoided. Assertions minimize the number of false negatives, because an assertion coupled with a call may signal an anomalous behavior that the trace cannot detect, and increase the likelihood of detecting mimicry attacks [244].

The most complex and rigorous strategy applies enriched traces to constrain both the system calls that P can issue, their ordering into traces and the values of variables. To support this strategy, PsycoTrace static tools analyze $SourceCode(P)$ to approximate the process self by returning a description of the $Self(P)$ that is $CFG(P)$ and $IT(P)$.

Definition ($CFG(P)$)**.** *$CFG(P)$ is a context-free grammar that defines the system call traces that P may issue during its execution.*

Definition ($IT(P)$)**.** *$IT(P)$ is an invariant table that includes a set of invariants $\{I(P,1),\dots,I(P,n)\}$, each associated with a program point i where P invokes a system call.*

This strategy merges the ability of constraining the sequence of system calls with that of associating memory assertions with such calls. This results in a high detection capability whose counterpart is the overhead due to the parsing of the trace and the evaluation of assertions. The thesis is focused on this strategy as it offers noticeable advantages. PsycoTrace can also support other strategies to define the process self that we have not investigated, such as a default-deny strategy that describes the process self through a specification language. A first example of this strategy is the one where the context-free grammar is a user input rather than an output of static tools. Obviously, this may increase the detection capabilities of PsycoTrace at the expense of a large number of false positives. This solution can be easily integrated into the framework to improve detection anytime the static tools return a description that does not constrain the behavior of P and it works even when $SourceCode(P)$ is not available.

1.2.1 Context-Free Grammar

To justify the description of the process self that has been adopted, consider that, even if in principle any behavior can be described just by associating assertions with system calls, we can simplify the description by formulating it in terms of sequences of system calls and of assertions coupled with system calls. A context-free grammar (CFG) is a synthetic description of the set of strings of system call tokens that the process can produce at run-time. We believe that a CFG is an acceptable compromise between two contrasting requirements, i.e. efficiency and complexity of the parsing. To check the consistency of the actual behavior of a process against the expected one, a run-time tool parses the string that describes the system calls produced up to a given instant and checks if it is a prefix of at least one grammar string. Hence, the overall run-time efficiency is inversely related to the parsing complexity and it is optimal for a regular grammar and less and less efficient for context-free and context-dependent grammars. On the other hand, as the grammar complexity decreases, it increases the probability of a false negative, e.g. of accepting a too large set of sequences and of classifying as normal an anomalous behavior. As an example, if a process invokes several system calls within distinct loops a regular grammar cannot constrain the traces when the two loops generate the same number of calls. Consider a process that opens some files, works on them and then closes all the open files:

```
for(i = 0; i < n; i++)
{
    ...
    open(file[i]);
    ...
}
...
for(i = 0; i < n; i++)
{
    ...
    close(file[i]);
    ...
}
```

A regular grammar cannot check that the same number of files are at first opened and then closed. Another advantage of CFGs is the ability of distinguishing among distinct invocations of the same system call in distinct program points. Furthermore, a CFG can describe the expected sequence of system calls and map calls with assertions, due to semantic actions, in a more concise and neat way than a regular grammar that cannot associate an action with a system call and a program counter (PC), but only with a system call. Lastly, several parser generator tools such as Bison [68] and the like takes as input grammar encoded in CFG syntax so that the adoption of a CFG simplifies the building of the system call parser.

Even if, in general, the trade-off between complexity and accuracy may depend upon the security level of interest, for the previous reasons, PsycoTrace adopts a grammar $CFG(P)$ to describe the legal traces of P, i.e. the sequences of system calls that P may produce. At run-time, the current trace of P, i.e. the sequence of system calls that the execution of P has generated up to a given instant, is legal if and only if it is coherent with $CFG(P)$, i.e. it is a prefix of at least one string of the language $L(P)$ generated by $CFG(P)$.

1.2.2 Assertion Generation

To compute assertions in enriched traces, the static tools apply a data-flow analysis that considers, for each system call s in $\mathit{SourceCode}(P)$, the set of reaching definitions of the instruction i that implements s. In general, s is a high level instruction that is implemented by invoking i. However, the detailed implementation of s is uninfluential as far as the computation of assertions is involved. A reaching definition for a variable v that may be referred by i is an instruction j that computes an expression e, assigns its value to v, and this value of v is still valid in i. If none of the variables that e refers to has been updated in-between j and i, then not only the value of v is still the one computed at j but also the assertion $v == e$ holds at i. Notice that, to generate assertions, we are interested in all the variables that reach i even if i does not refer all of them. This is the reason why the notion of reaching definitions of PsycoTrace generalizes the classical definition by saying that v is any variable that i may refer, rather than the variable that is actually referred by i. If several definitions for v reach i and none of their variables has been updated in-between j and i, then the value of v is equal to the expression in one of the definitions that may reach i. Hence, an assertion:

$$(v == e_1)|(v == e_2)|\dots|(v == e_k)$$

holds at i, where $j = 1, \dots k$ is an instruction that computes e_j and assigns its value to v.

The cases previously defined can be generalized if we consider that PsycoTrace run-time tools can access the process variables through introspection anytime it invokes a system call. By exploiting this ability, PsycoTrace can define an assertion even if a variable w in e has been updated in-between j and i because run-time tools may access and copy the value of the variable w of P before it is updated. As an example, an assertion can be generated anytime there is at least one system call in-between j and the first statement that updates w because, when analyzing this call, the run-time tools can save the current value of w. In this case, the assertion coupled with a system call refers to the saved value rather than to that in the memory of P. If i refers several variables $v_1, \ldots, v_j$ then the assertion coupled with i has the structure:

$$A(v_1)\& \ldots \& A(v_j)$$

where each $A(v_h), h = 1, \ldots, j$ is computed as previously defined.

In the most general case, invariants coupled with system calls of enriched traces are based upon reaching definitions for any program variable. Instead, if some other PsycoTrace strategies to describe the process self are adopted, then the static analysis may consider the reaching definitions of system call parameters only.

1.3 Formal Models for Static Analysis

This section presents a more formal description of the previous analysis in terms of a reaching definitions analysis. Then, it describes an alternative approach to compute assertions based upon an abstract interpretation framework.

1.3.1 Reaching Definition Model

The set of reaching definitions $r(q)$ of a program point q is defined as the set of pairs (x, p) where x is a program variable and p is a program point where the definition of x computed at p may reach q. Suppose q is an assignment, and:

- $expr(q)$ is the right-hand side of the assignment;
- $var(q)$ is the left-hand side of the assignment.

To compute an assertion coupled with q, let us consider a variable x and suppose, for the moment being, that there is just one definition for x that reaches q, i.e. $r(q) = (x, p)$, then we may map with q the assertion $(var(p) == expr(p))$ if the following condition holds:

$$\forall y \in expr(p), \quad (y, s) \in r(p) \Leftrightarrow (y, s) \in r(q)$$

Informally, if any definition that reaches p also reaches q, then the value of $expr(p)$ does not change if it is computed at q. Hence, the corresponding assertion holds at, and may be coupled with, q. This condition is sufficient but not necessary since one ore more statements can update a variable in $exp(q)$ in-between p and q but without changing its value. If the previous condition is not satisfied because some variables in $expr(p)$ are updated, then we may generate an assertion by saving a copy of the values of these variables. Thus, if z is the first system call instruction after q and the following condition holds:

$$\forall y \in expr(q), \quad (y, s) \in r(p) \Leftrightarrow (y, s) \in r(z)$$

then the value of variables of interest has not be updated and we can read the values of some variables in z and the assertion in q may refer to the values that have been copied in z rather than the current ones in the memory of P.

1.3.2 Abstract Interpretation Model

This section describes how to transform any abstract interpretation into one that returns the assertions coupled with a program point.

An abstract interpretation of a program simulates its execution by modeling the effects of every statement on an abstract machine. If the behavior of the language constructs is properly defined, then the abstract execution over-approximates the behavior of the system. This implies that the abstract system is simpler to analyze, but it may lack completeness, i.e. not every property true in the original system holds for the abstract system. On the other hand, abstract interpretation is sound, i.e. every property that is true in the abstract system can be mapped onto a true property in the original system. Several abstract interpretations have been defined that formally define an analysis that returns linear relations among variables of programs. One of the main uses of these relationships is to compute at compile time a specified numeric sub-range for each integer variable so that each integer expression may be coupled with a range that always includes the actual value of the expression itself. In this way, each abstract interpretation maps with each program point, i.e. each arc of the control-flow graph, an assertion that states that each variable belongs to the concretization of the sub-range.

While each of these interpretations can be applied to compute assertions, we show how to transform each interpretation into one that fully exploits introspection to increase the accuracy of assertions and, hence, the amount of information that is available on the process self. Any abstract interpretation consists of:

- an abstract domain $\alpha(S)$, which maps concretes objects (states, traces, ...) with abstract ones;
- a set of abstract operators *AbsOp*, to model the effects of the language constructs on the objects of the abstract domain;
- the abstraction function α, to map concrete objects into abstract ones;
- the inverse concretization function γ, which maps an abstract object into some concrete one.

Often it is required that (α, γ) is a Galois connection. Any abstract interpretation maps each program point with an abstract state $s \in S$ that maps each program variable into an abstract value.

To transform the original abstract interpretation into the one to generate assertions, first of all we consider a set of program points and map each point in this subset with a distinct label $l \in Lab$, where Lab is a finite set of labels. Then, we define an abstract state of the new interpretation as a finite set of elements from the Cartesian product of the abstract state domain of the original interpretation and Lab. Hence, a new abstract state s_1 consists of a set of pairs $S_1 = \{< s \in S, l \subseteq Lab >\}$, where s is defined as in the original abstract interpretation. A new abstract state cannot include two pairs such that the second elements of the pairs are equal. In this case, the two pairs are merged into a pair where the first element is the least upper bound of the two states in the original domain and the second element is equal to the second element of the two original pairs. In the new abstract interpretation, if a program point receives an abstract state s_1, then it computes an abstract state that includes all the pairs $< os, ol >$ that can be computed starting from any pair $< s, l >$ in S_1 as follows:

- os is produced by applying the abstract operators in $AbsOp$ to s,
- ol is produced by inserting the label (if any) coupled with the considered point into l.

A program point where two or more control-flows merge produces a single abstract state that merges the abstract sets coupled with each flow, whereas a program point with two or more possible successors transmits the abstract state that it computes to each successor. To prove that we have defined an abstract interpretation, consider that Lab is finite and:

(a) if there are no loops, the number of states that reach a program point depends on the number of control-flow paths;

(b) if there are loops, the fixed point of the abstract state coupled with a program point is reached when the abstract version of the language constructs add no further labels to the abstract states that reach the point. This fixed point is reached since Lab is finite.

The new abstract interpretation maps each program point with a set of states that depend upon the instructions belonging to the paths that reach the program point. Moreover, it does not consider the number of times an instruction has been executed, but only the distinct paths of the control-flow graph to the instruction. Then, we may map each program point with an assertion generated as follows: given the abstract value of a variable, the assertion states that the concrete value of this variable should belong to the set that is the union of the concretizations of the values in the set that includes all the abstract values of the variable in the different elements of the abstract state. In other words, we consider all the values of the variable in all the pairs in the abstract states, map each value into the concrete ones and merge all these concrete values. As an example, if an abstract state includes n pairs that map x into, respectively, $a_1, \dots a_n$, then the assertion states that x belongs to the union of the concretization of a_i, $1 \leq i \leq n$.

To return assertions that constrain in a more accurate way variable values, PsycoTrace run-time tools can remember those labels that have been met during the current execution of P. In this way, when building the set of abstract values for a variable, run-time tools can include only those states coupled with labels that have been met in the current execution. At the expense of analyzing at run-time some abstract information coupled with a program point, namely the labels coupled with each element of an abstract set, this strategy reduces the amount of indeterminacy due to a static analysis. This also shows the difference of defining abstract interpretation for program monitoring and for program optimization because the latter, in general, *does not access the actual program state.* In our framework, the only instructions that may be coupled with a label are system calls and the abstract states of interest to deduce assertions are those coupled with a system call.

A distinct approach exploits the notion of system block graph (discussed in Chap. 7) and introspection to increase the precision of the abstract interpretation, by reducing the complexity of the code to be statically analyzed. In fact, by analyzing each system block, i.e. the code in-between two successive OS invocations in isolation, this approach computes the relations between the variable values at the beginning and those at the end of the system block. By exploiting these relations, and the ability of applying introspection to access the values of these variables at the beginning of a system block, this approach strongly increases the amount of information on variable values at the end of a system block and, hence, the constrain on the process behavior that the static analysis can return. This strategy not only increases the overall precision of this approach but also reduces the complexity of the static analysis because it considers only a set of code fragments, i.e. the system blocks, each analyzed independently from the other ones, rather than a complete program.

In the next chapters, we first discuss virtualization, because it allows PsycoTrace run-time tools to implement introspection in a transparent way, and then, after reviewing some related works, we will discuss in details each component of PsycoTrace and the corresponding implementation.

Chapter 2

Virtualization-based Security

This section discusses how virtualization can increase the overall security level of an ICT system by simplifying attack detection and confinement. In fact, while introspection simplifies the analysis of the state of a virtual machine to detect attacks, the ability of increasing the number of virtual machines running user applications and services minimizes the sharing among such applications and services. A key advantage of virtualization is that its adoption may be transparent for the applications and the OSes. It is important to notice that the most popular reasons that favor the adoption of this technology are related to rather distinct considerations, such as the strong simplification in system administration and management as well as to energy saving. Hence, the ability of exploiting the technology to increase the overall security is a further important advantage that should not be missed.

2.1 Virtualization Technology

In computer science, a virtual machine (VM) is an abstract computing system that is defined not to build a physical machine but as a step towards the solution of a problem. As an example, any computing system can be described as a hierarchy of VMs that covers the range from an actual physical machine to the abstract one that interfaces the final user. Any machine in the hierarchy is built on top of, and it abstracts, the underlying machine to define programming languages and resources more oriented to the problem of interest. This design and implementation strategy simplifies the portability and the reuse of one or more layers, as exemplified by the Java virtual machine. Here we will consider a more specialized version of the VM concept and of the corresponding technology because we are interested in VMs that implement a *software emulation* of the program environment defined by a physical architecture. Such a VM enables a user to run the program stack, from the OS to the applications of interest, as it happens on a physical architecture. Properties of the architecture to be emulated, such as the amount of memory, or the connected physical devices are fixed when the VM is configured. The replacement of a physical system with a VM should be fully transparent to the applications and to the OS, so that they can run unchanged.

While virtualization is rooted in time-sharing and in IBM OSes [110, 175, 211], interest has been revitalized by the diffusion of personal computers, as an attempt to preserve their existence in spite of Moore's law because the increasing computing capability of hardware components reduces their utilization and, hence, their cost effectiveness.

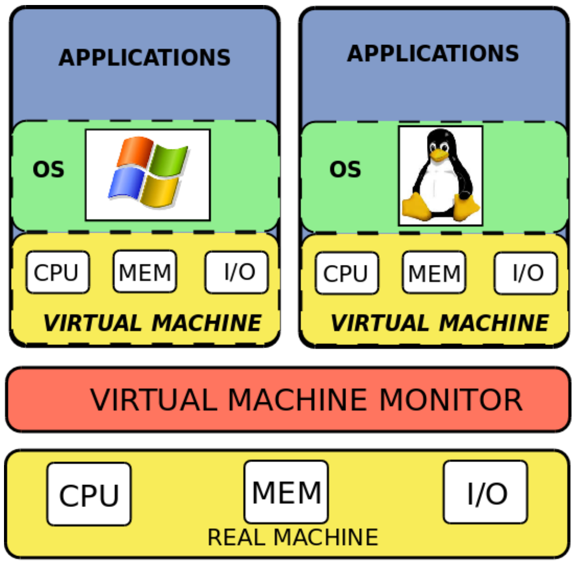


Figure 2.1: Type I Virtual Machine Monitor

If we consider the hierarchy of software layers proper of any computer system, there are two of these layers where our flavor of virtualization can be introduced. To define them, first of all we introduce the concept of *virtual machine monitor* (VMM) [104, 103], the software component that creates, manages and monitors VMs. Two kinds of VMM exist:

Type I VMM: this VMM is a thin software layer that runs on top of the hardware/firmware layer. The VMs on this VMM emulate the behavior of the underlying physical machine. In this way, a standard machine can support several VMs, each running a distinct OS, as shown in Fig. 2.1.

Type II VMM: this VMM runs on top of a *host OS* and a VM is implemented as a process that runs an emulator of the physical architecture of interest. Each VM supports a *guest OS* that, in turn, supports user applications (see Fig. 2.2). Resource requests from an application are transmitted to the guest OS that maps them into requests to the host OS of the underlying physical machine. In general, the resources that the host OS allocates to a VM are those that can be accessed by the user that creates the VM. The VMM determines in either a dynamic way or at configuration time, e.g. when the VM is created, the amount of resources to be allocated to each VM.

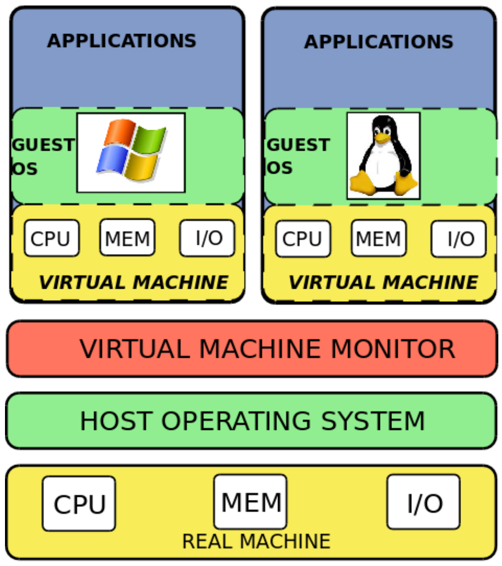


Figure 2.2: Type II Virtual Machine Monitor

Each type of VMM can emulate a physical machine by applying a range of implementation strategies that span from *run-time interpretation* of each assembly instruction to *dynamic rewriting* of sequence of assembly instructions by the emulator that then stores the output of the translation in a cache to minimize the translation overhead. Assuming that a VM has the same interface of the physical machine, then any implementation strategy should maximize the number of instructions directly executed by the physical machine without any software mediation. To determine instructions that can be directly executed, first of all we have to consider *sensitive instructions*, i.e. those assembly instructions, such as I/O ones, whose execution involves the resources shared with other VMs and that can reveal that the physical machine has been replaced by a virtual machine. The execution of sensitive instructions depends upon the mapping of virtual resources into physical ones and it cannot be fully implemented by dynamic rewriting only. Hence, any implementation strategy can, at most, avoid any software mediation for non-sensitive instructions only. A simple condition to verify if an architecture can be virtualized is if *it allows any sensitive instructions to be trapped when executed in any but the most privileged mode* [187]. This is the reason why several systems run the assembly code of a VM in a low privilege ring so that any attempt to access a physical resource by a sensitive instruction results in an exception. The handler of the exception can resume the emulator that is in charge of managing the physical resources. While the solutions proposed by this thesis can be applied to any VMM, in the following, because of performance and security reasons, we will consider solutions based upon a type I VMM only. First of all, the existence of two OS layers strongly increases the complexity of the execution of sensitive instructions. Furthermore, the complexity of these layers increases the number of vulnerabilities and reduces the overall robustness. Our choice in favor of a type I VMM because of its low complexity with respect to an OS has the important implication that any solution to increase the overall robustness should minimize the number of extensions to the VMM.

Since one of its goals is to confine erroneous or malicious behavior of any VM, the VMM should monitor any access of a VM to shared resources such as primary or secondary memory. Furthermore, it should also prevent a VM from exhausting a shared resource because this may slow down other VMs. The confinement that a VMM can guarantee is fundamental for security because it strongly increases the complexity of attacking another VM even for an attacker that already controls a VM. This is the reason why VMs may be used to analyze code that is potentially dangerous or to debug a system.

2.1.1 Current Products

From our point of view, the most interesting VMMs currently available are those supplied by, respectively, VMware [119] and Xen [26].

2.1.1.1 VMware

VMware VMMs can be stand-alone or hosted. A *stand-alone* VMM is the one more interesting for our approach as it is basically a type I VMM that runs directly on the hardware and that lets users create their VMs. On the contrary, a *hosted* VMM is a type II VMM that runs as an application on a host OS that it exploits for memory management, processor scheduling, hardware drivers, and resource management. Since VMware products are targeted towards x86-based workstations and servers, they have to manage the problems posed by a not fully-virtualizable architecture. As a matter of fact, the previous condition on sensitive instruction is violated by the x86 that contains both non-privileged sensitive instructions and privileged instructions that fail silently [197]. To this purpose, portions of the code of a VM have to be dynamically rewritten to insert traps wherever the VMM intervention is required. To minimize the resulting overhead, the translation results are cached and reused wherever possible. This solution is strongly related to the x86 architecture, where the protection mechanism provides four privilege levels, or rings, from 0 through 3. In the original design, ring 0 is meant for OSes and kernel services, ring 1 and 2 for device drivers, and ring 3 for applications. However, in most cases, both the OS and the device drivers run completely in ring 0 and applications in ring 3. Privileged instructions may be executed only in ring 0, and cause a protection violation if executed in any other ring.

VMware Workstation is a hosted VMM that has three components: the VMX driver and VMM installed in ring 0, and the VMware application (VMApp) in ring 3. The VMX driver is installed within the OS to gain the high privilege levels required by the VMM. When it is executed, the VMApp cooperates with the VMX driver to load the VMM into kernel memory and assign to it the highest privilege ring, ring 0. At this point, the host OS knows about the VMX driver and the VMApp but not about the VMM. Now the machine supports two worlds: the host world and the VMM world. The latter interacts directly with the processor hardware, or through the VMX driver, with the host world. However, every switch to the host world is expensive from a performance perspective as it requires all the hardware states to be saved and restored on return. When the guest OS or an application run CPU-bound programs, they are executed directly through the VMM. Instead, I/O instructions are privileged ones that have to be trapped by the VMM and executed in the host world. I/O intensive applications are slowed down because the I/O operations requested by a VM are translated into high-level I/O-related calls. These calls are eventually invoked through the VMApp in the host world and their results are communicated back to the VMM world.

The ESX server is a standalone VMM that does not require a host OS and can run on a bare machine. It handles all the I/O instructions, which require the installation of all the hardware drivers and related software, and it implements shadow versions of system structures, such as page tables (see Fig. 2.3), by trapping every instruction that attempts to update them. This corresponds to the adoption of one extra level of mapping in the page table. The virtual pages of a process are mapped into physical pages through the page tables of the guest OS. Then, the VMM maps a physical page into a machine page that, eventually, is the correct one in the physical memory. The ESX server applies several techniques to increase the overall efficiency and levels of isolation to keep VMs independent from one another.

2.1.1.2 Xen

Due to its inherent features that cannot be virtualized [197], the x86 architecture increases the complexity of achieving both high performance and strong confinement. In addition, completely hiding the effects of resource virtualization from guest OSes may result in several problems for both correctness and efficiency. These reasons underlie the choice of some researchers at University of Cambridge to design and develop Xen, a modified architecture for virtualization that exports a *para-virtualized* architecture to each of its VMs to maximize performance and resource isolation while preserving the same application binary interface as commodity OSes. While full virtualization exports to the OS an exact replica of the interface of the underlying architecture, para-virtualization requires some modifications of the OS to be run on a VM (see Fig. 2.4). Although Xen requires the porting of an OS, the minimization of this effort is a project goal. Another goal is running hundred of VM instances on a single physical machine with a reasonable

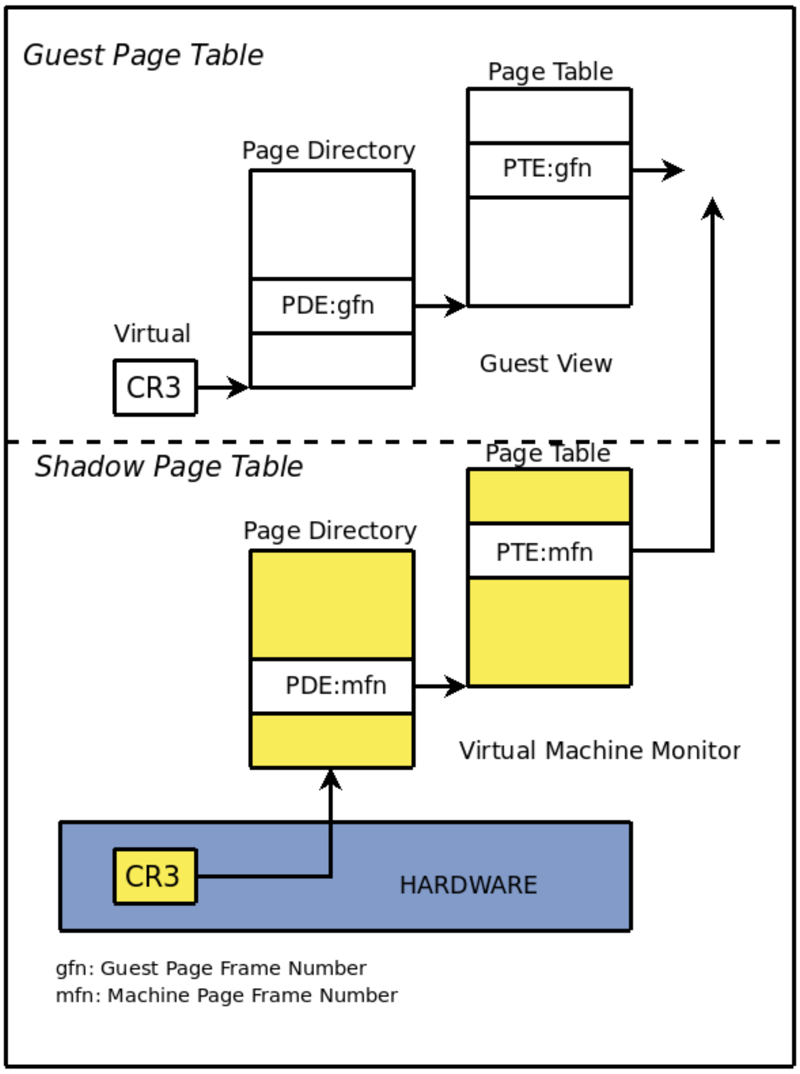


Figure 2.3: Shadow Page Table

performance. By applying para-virtualization, Xen can export a new VM interface that aims at improving both performance and scalability. A new lightweight event mechanism replaces the traditional hardware interrupts in the x86 architecture for both CPU and device I/O. Asynchronous I/O rings (see Fig. 2.5) are used for simple and efficient data transfers between the VMs and the VMM, or *hypervisor* in Xen terminology. For security purposes, each VM registers with Xen descriptor tables for exception handlers and, with the exception of page faults, the handlers remain the same. To avoid the indirection through Xen on every call, guest OS can install fast handler for system calls, allowing direct calls from an application into its OS. Even if guest OSes have a direct access to hardware page tables, to implement a secure but efficient memory management technique, page table updates are batched and validated by Xen. Instead, in VMware systems, the VMM traps and applies

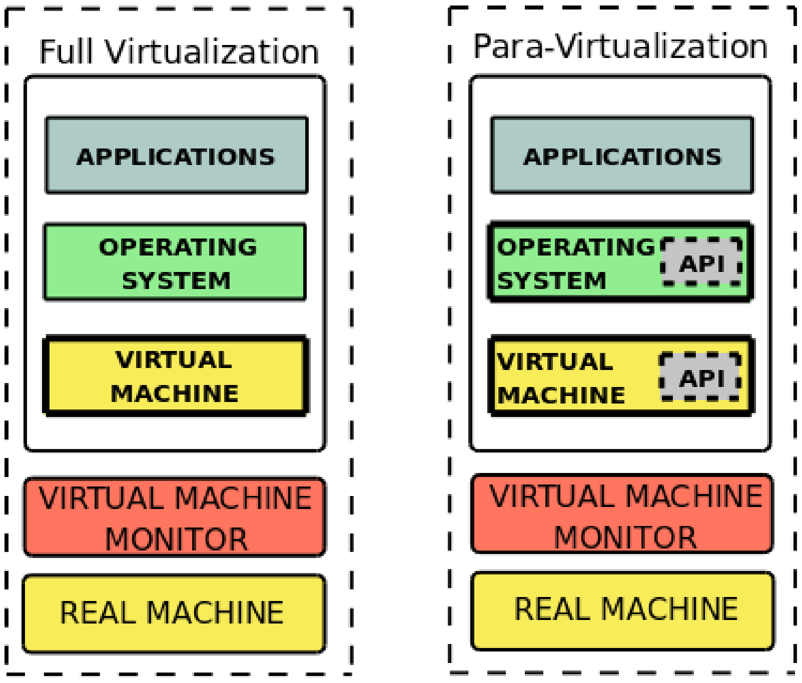


Figure 2.4: Full Virtualization and Para-Virtualization

every update to the page table. In particular, the invocation of a `fork()` to create a process, results in a huge number of updates that might result in a noticeable performance loss that can be reduced through Xen batched updates. Each guest OS can access a timer interface and is aware of both "real" and "virtual" time. In this way, Xen tries to build a more robust architecture that preserves all the critical features for application binaries while minimizing the porting effort for a guest OS.

Xen Memory Management. A first consequence of para-virtualization is that each time a guest OS updates the memory mapping of a process, Xen has to intercept the update to prevent interferences among VMs. To deal with memory virtualization, one the most complex task for a hardware-level VMM, Xen considers three distinct issues:

1. physical memory management, e.g. how to avoid memory fragmentation;
2. virtual memory management, e.g. how to minimize the overhead introduced by VMs scheduling;
3. page table (PT) management, e.g. how to validate each memory access to satisfy the isolation requirement among VMs.

To give to guest OSes the illusion of a contiguous address space, Xen defines two distinct address spaces: Machine memory, i.e. the total amount of physical

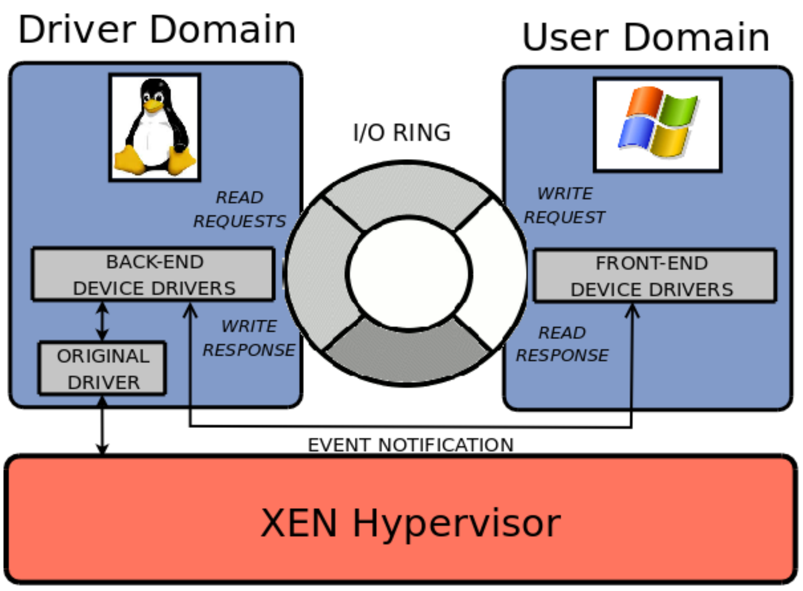


Figure 2.5: Xen Split Device Drivers

memory of the host that runs Xen, and Pseudo-Physical memory, i.e. the space address as seen inside a VM. Two tables implement the mapping between the two address spaces: Machine-to-Physical (M2P), which maps physical memory pages into pseudo-physical pages, and Physical-to-Machine (P2M), one for each domain, which implements the reverse mapping. The size of M2P is proportional to the physical memory, whereas that of a P2M is proportional to the memory allocated to each VM. To minimize the performance degradation of VM context switching due to TLB misses, the topmost 64MB (for 32 bit architecture) of the virtual address space of each process records a mapping for the Xen hypervisor itself.

There are two possible solutions to manage PTs: shadow PTs or direct management of the PTs by guest OSes. Shadow PTs require that a guest OS implements virtual PTs that are not visible to the MMU. In this case, to prevent interferences among VMs, Xen traps each access to the virtual PTs and propagates their updates to the real PTs used by the MMU. Direct management of PTs requires that guest OS PTs are read-only so that the OS is forced to invoke Xen through hypercalls to update the mapping.

2.1.2 Hardware Support

While the inherent lack of support increases the complexity of virtualizing the x86 architecture, Intel and AMD have recently implemented some processor extensions to make this architecture classically virtualizable. Intel virtualization technology (VT-x) and AMD extensions (AMD-V) are not completely equivalent but they share the same basic structure. Intel VT-x introduces new modes of CPU operation: *VMX* *root operation* and *VMX non-root operation*. VMX root operation is a host mode similar to previous IA-32 operation before VT-x and is intended for VMMs, while VMX non-root operation is a guest mode targeted at VMs. Both modes support execution in all four privilege rings. The `VMRUN` instruction performs a VM Entry, switching from host to guest mode. The inverse switch occurs on a VM Exit that may be triggered by both conditional and unconditional events. For example, a write to a register or memory location might trigger such a transfer according to the bits that are modified.

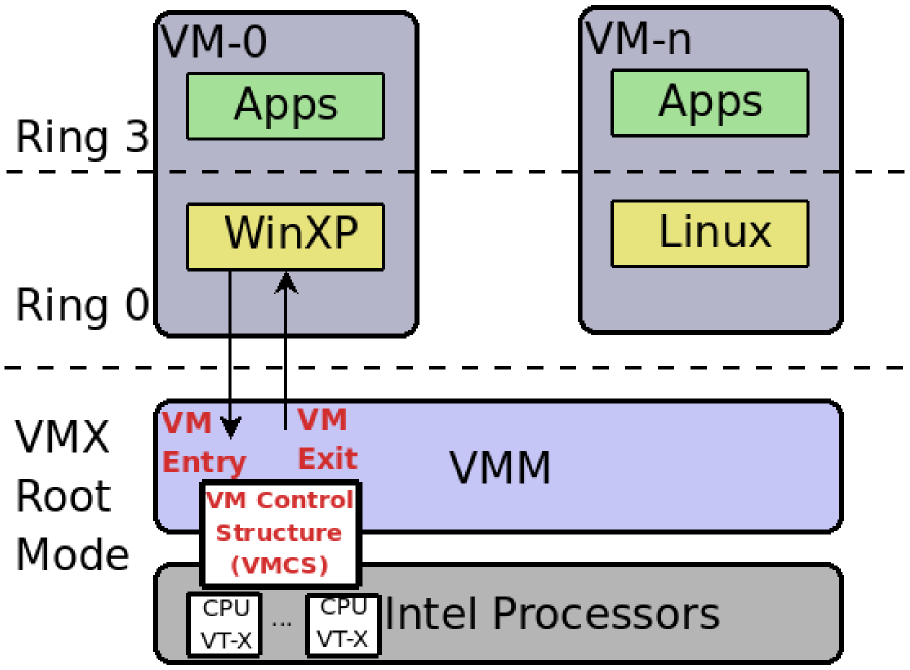


Figure 2.6: VM Exit and VM Entry Operations

The interaction between hosts and guests exploits the VM control structure (VMCS) that records the guest state and the host state. On a VM Entry, the host processor state is saved before loading the guest processor state from the VMCS. In a VM Exit, these operations are swapped: at first the guest state is saved and then the host state is loaded (see Fig. 2.6). The processor state includes segment registers, the control register 3 (`CR3`), which stores the physical location of the page tables, and the interrupt descriptor table register. The address space of a guest VM can be separated from that of the VMM by loading and storing `CR3` on VM Entry and Exit. To speed up VM Entry and Exit, the VMCS does not store general purpose registers as the VMM can include them as needed. Furthermore, the VMCS of a guest is referenced through a physical address to avoid translating a guest virtual address. The most important difference between host and guest mode, VMX root and non-root operation, is that several instructions in guest mode will trigger a VM Exit as specified by the VM's execution control fields. Among the conditions that trigger this exit we have:

- external-interrupt or interrupt-window exiting,
- access to the task priority register in the VMCS and CR masks and shadows,
- exception, I/0 , and access to specific registers.

To quickly identify the problem and return the control to the guest VM, a VM Exit also includes information on the reasons for the exit. An event is a two-way communication channel with event injection that enables the VMM not only to receive events from a guest but also to delegate the management of interrupts or exceptions to a guest VM using the interrupt descriptor table (IDT). By introducing a new execution mode with full access to all four privilege rings, both the ring compression and the ring aliasing problems disappear [152]. A guest OS executes in ring 0 while the VMM is still fully protected from any errant behavior. Since each guest VMCS is referenced with a physical address and it stores critical registers, VMs have full access to their entire address space. Moreover, the VT-x supports a fine-grained control over any potentially problematic instruction. Lastly, the VMCS control fields also address the challenge of interrupt virtualization. External interrupts can be set to always cause a VM Exit, and VM Exits can be conditionally triggered upon guest masking and unmasking of interrupts. In this way, the x86 can becomes classically virtualizable. VT-x strongly simplifies the VMM with respect to both para-virtualization and binary translation. However, the overall performance may not be fully satisfactory because, sometimes, a software VMM may achieve a better performance due to the high cost of managing shadow pages to handle page faults. Furthermore, the VMM has to determine the cause for a VM Exit from the VMCS.

2.1.3 Transparency

A condition for the transparency of a virtualization-based security approach is VM transparency. A VM is transparent if its presence cannot be detected by the software that it supports so that any test that this software can implement should return the same result if executed on a VM or on a physical architecture. While it is well known how to achieve transparency in the case of functional tests that only consider the input/output behavior of a program, transparency is rather more complex if the test considers non functional properties such as the amount of resources a program accesses or timing-based properties [92]. An example is a test where the output is the number of cache faults or the program execution time. However, the problem is simplified because most of these tests have not a single result even when applied to a physical architecture. Taking this non-determinism into account, the existence of a VM cannot be detected if any result returned by a test on a VM belongs to the set of results returned by at least one physical machine.

This implies that a program can implement tests that detect whether the execution environment has changed with respect to a predefined one but these tests cannot distinguish whether distinct results are due to the adoption of a VM or of a distinct physical machine. As an example, the number of cache faults changes not only when moving from a physical machine to a virtual one but also when moving to a distinct physical machine or when the number of user programs changes and so on. The impossibility of detecting a VM under very general assumption has been proved in [109].

In this thesis, we are not interested in building transparent VMs, i.e. VMs that are "undetectable" by an attacker. In our view, we are more interested with the transparency enabled by virtualization technology that is the feasibility of checking the integrity of the software that a VM runs both at the kernel-level and at the user-level *without modifying this software or requiring some user intervention.* This implies that the integrity of the overall status of the system hosted by the VM can be measured, i.e. checked, without forcing the users to install further tools or to modify any software or architectural layer. This transparency is achieved by implementing an access to each virtual component of the monitored VM, e.g. to its main memory or the processor's register, to read and modify their current state. As we will discuss later, this access makes it possible to check the consistency of any running process without modifying the software of the monitored VM and without the users of this VM being aware of these checks. It is worth noticing that transparency is rather important from a security point of view because the more transparent a solution, the more complex for an attacker to discover (and to attack) the components that implement the controls. As an example, if a solution is fully transparent to the OS, then a malicious user cannot discover the monitoring even if she can access the OS.

2.1.3.1 Security and Transparency

From a security perspective, virtualization introduces a further system level, the VMM, that can independently access any value in the state of a VM. Hence, it can behave as a coprocessor that can access both the memory and the CPU status of a physical machine to implement checks that may involve not only variable values but also the process control-flow. The evasion of these checks is very complex due to the low-level access of the VMM to the memory representation of any components. From a semantic point of view, we have modeled these checks as the evaluation of an assertion on the status of the component. As a counterpart, there is the cost of developing this new layer and the potential performance loss. The complexity of the new layer is large because the VMM should access a process memory on a VM to evaluate assertions that have a complexity that strongly depends upon the required security level. This contrasts with the assumption that the VMM should be rather simple to minimize its vulnerabilities. A solution can be found by shifting this complexity to one VM that the VMM supports. This strategy recalls to the one that Xen adopts to simplify the implementation of physical I/O.

Rather than implementing the I/O drivers in the Xen VMM, a privileged VM runs these drivers, whereas the drivers of virtual I/O devices run on other VMs and interact with those on the privileged VM. In our considered problem, a privileged VM evaluates the invariants while the VMM only needs to implement those functions that enable the privileged VM to access the variables in the memory of another VM that are referred by the invariants. The resulting strategy will be denoted as virtual memory introspection (VMI) [95] and it involves two VMs, the *monitored VM* and the *monitoring VM* (which we also refer to as *privileged VM* or *introspection VM*). An implementation exploiting two distinct VMs is very robust because of the confinement that the VMM implements so that even an undetected attack against the monitored VM has a low probability of spreading to the monitoring VM.

A further problem to consider is the frequency to evaluate an invariant or, from a program point of view, which instructions of the monitored process may be coupled with an invariant. We have already discussed the reasons in favor of monitoring the system calls that a process issues. For this reasons, the status of a VM should be checked any time one of its processes invokes a system call but the VMM can implement the checks only when the implementation of the system call invokes one of its function. To evaluate invariants with the required granularity, all the system calls in the monitored VM should be trapped and control transferred to the privileged VM.

The two main strategies to trap system calls in a VM require to hijack, respectively, the system call table and the memory mapping. The first solution updates the system system call table in the monitored VM so that the invocation of a system call transfers control to a wrapper that suspends, e.g. freezes, the VM and transfers the control to the VMM and then to the monitoring VM. This solution is rather efficient because it minimizes the overhead due to the cooperation between the two VMs. The approach is fully transparent to the applications only but not to the OS because of the updated system call table and the wrapper code.

The second solution inserts a trap into the emulator to discover when some virtual memory positions are accessed. The trap transfers the control to the emulator anytime some predefined positions in the memory of the VM, namely those involved in the execution of a system call, are accessed. Again, the code in the emulator will freeze the VM and transfer the control to the monitored VM. This solution is fully transparent to both the application and the OS but it requires the update of the emulator code.

2.2 Virtual Machine Introspection

As shown in Fig. 2.7, *Virtual Machine Introspection* (VMI) enables a privileged VM, or Introspection VM (I-VM), to retrieve critical data structures in the memory of a Monitored VM (Mon-VM) to evaluate a set of invariants on data-structures at the kernel or at the user-level. In this way, an I-VM can analyze the state of the processes and of the kernel hosted on a Mon-VM at the hardware/firmware level, without introducing additional units. Hence, introspection is applied at a lower level than the one an attacker can gain and it is very hard to elude it. Thus, the advantages of VMI are:

1. *full visibility* of the system running inside the Mon-VM, because the I-VM can access every Mon-VM component, such as the main memory or the processor's registers;
2. *more robustness*, because the I-VM is isolated from the Mon-VM;
3. *transparency*, because the security checks are implemented without requiring any update of the software running in the Mon-VM and are almost invisible.

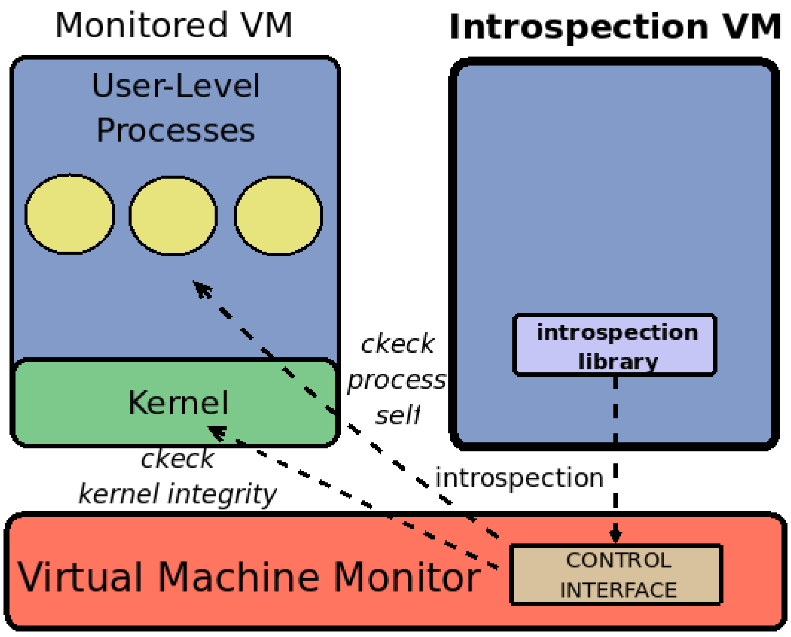


Figure 2.7: Introspection Virtual Machine

To exemplify the various invariants that can be evaluated consider that the I-VM can compute the hash of the code of any running software module and compare it against a value computed offline to discover whether the module has been maliciously updated. If applied at the kernel-level, this approach supports the discovery of rootkits, while at the user-level it supports the detection of attacks to the process self. To prove both the power and the precision of VMI consider that further invariants may be evaluated:

1. to guarantee the integrity of critical kernel data structures;
2. to assure that a process correctly executes the application code.

Alternative implementations of VMI exist. VMI is wholly *passive* if the VMM only provides mechanisms for the I-VM to peer into another, actively running Mon-VM. In a distinct solution, the VMM can support an event-based mechanism, i.e. *trigger-based*, where the I-VM is notified when certain events occur. Other differences arise because the event notification may be *synchronous*, i.e. the Mon-VM is stopped during event processing, or *asynchronous*, i.e. the Mon-VM is running as the event is delivered. We briefly review the advantages and disadvantages of these alternative solutions.

2.2.1 Passive Virtual Machine Introspection

A passive VMI-based system monitors the status of the Mon-VMs without actually interrupting them, or by minimizing the interruption time. In a passive environment, the I-VM periodically examines the memory and the critical CPU registers of the Mon-VMs to detect unexpected changes or settings. As an example, the I-VM maintains a shadow copy of the IDT and periodically compares it against the one that the Mon-VM currently uses. The advantage of passive VMI is its minimal performance impact on the Mon-VMs. A first disadvantage is due to the periodic nature of the checks that results in a delay in detecting changes to the Mon-VMs. From this point of view, it is important that the frequency of the checking be unguessable, e.g. the time-window in-between two checks should be random. If not, malicious code in the Mon-VMs can potentially predict when the check will be performed and undo the change during that time interval. This would prevent the I-VM from detecting the change. Another disadvantage is the large number of changes that can be made during the time-window. The cumulative effect of these changes could increase the overall complexity of undoing them.

A major challenge of this solution is preserving the semantic consistency of the data structure of the Mon-VM, since these data structures are concurrently accessed by both the Mon-VM and the I-VM. Therefore, the values that the I-VM reads may not be consistent and, if this issue is neglected, erroneous results could be produced. A recovery-based solution that undoes unauthorized changes is also a challenge when the Mon-VM CPUs continue to execute instructions. In fact, the recovery code needs to ensure that the changes to the Mon-VM does not affect the consistency of the status of this VM that, otherwise, will likely crash.

2.2.2 Trigger-Based Virtual Machine Introspection

In an alternative implementation, the I-VM sets triggers that are activated when a specified condition occurs. For example, any write to a particular memory address activates a trigger [242]. In a trigger-based system, the I-VM sets triggers to monitor writes to critical data structures. For example, any attempt to write to the memory region storing the IDT activates a trigger, and eventually notifies the I-VM of the change or of the attempted change. Trigger-based VMI removes the delay in detecting unauthorized changes that is inherent in passive VMI. Furthermore, since each individual change activates a trigger, the increase of complexity due to cumulative memory changes may be resolved.

The delivery of the trigger can either be asynchronous or synchronous. In the former case, the Mon-VM continues to execute instructions, in the latter the Mon-VM is stopped while the I-VM processes the trigger. Also hybrid solutions are possible, where only the CPU that activates the trigger is stopped while other CPUs are unaffected. The performance impact on Mon-VMs of asynchronous delivery of triggers is similar to that of passive VMI. However, because the I-VM is executed only to handle a trigger action, the performance impact on the overall system is proportional to the frequency of trigger actions. Asynchronous delivery does, however, share with passive VMI the challenge of preserving memory consistency.

Synchronous delivery of triggers freezes the Mon-VM CPUs and this necessarily has the largest impact on these VMs. Again, the impact is a function of the frequency of the activation of triggers. As an example, if the I-VM sets write triggers for the IDT, and no attempts to write to the IDT ever occur, then the overhead is quite low. However, if the granularity of triggers is at the page-level rather than at the word-level, then spurious triggers may be activated due to writes in other words in the same page and the resulting overhead may be rather high. Controlling the performance impact on Mon-VMs with synchronous delivery of triggers is a primary concern that may be accomplished by applying other techniques. For example, if the introspection API provides a periodic timer, then techniques similar to passive VMI can be applied.

2.2.3 Mitigation of Threats

VMI offers several advantages in security monitoring and control. Scenarios where VMI can provide significant advantages include:

- intrusion detection/prevention: by providing at least some of the monitoring functionality from “outside” the Mon-VM, the I-VM can be immune to modifications to the compromised Mon-VM made by an intruder. One of the axioms of computer security is that once the attacker gains administrator privileges in the monitored system, any IDS can no longer be trusted. VMI enables the I-VM to continue unharmed even in this case;

- malware detection and control: similar arguments hold for malware detection and control. If a malware has affected a Mon-VM and, even, has made it unavailable, an I-VM can continue to monitor the operations of the Mon-VM;

- distributed attacks or reconnaissance activities: they may be sufficiently low-profile to go undetected by individual machines. However, if an I-VM controls several Mon-VMs, then it might be able to correlate these activities into a significant pattern.

2.2.4 Vulnerabilities

Although powerful, VMI increases the overall complexity and, in turn, introduces additional vulnerabilities due to flaws in the design, implementation or configuration of:

- the introspection mechanisms that allow the Mon-VM to modify, hide or build fake information about its activities;
- the I-VM that allow the Mon-VM:
  - to crash the I-VM, whereas the execution of the Mon-VM continues instead of being stopped or of producing an alert;
  - to manipulate the I-VM, for example by crafting contents of memory to trigger a buffer overflow in the I-VM. This would allow an attacker to violate the isolation properties of VMI and, potentially, to directly control the I-VM.

The risks due to these vulnerabilities can be mitigated by state-of-the-art careful design of the involved components, including clean separation between components, clear communication and control paths, separation/limitation of privileges, and careful coding practices.

2.3 Applications of Virtualization to Security

The application of virtualization enables PsycoTrace to implement introspection in a transparent and robust way. Figure 2.8 shows the resulting overall architecture of PsycoTrace, where the run-time components receive as input the description of the process self and apply VMI to check kernel integrity and the process self (see Chap. 5 for a detailed description of the run-time architecture).

In the following sections we discuss further applications of virtualization that can enhance the security of computer systems, namely to check kernel integrity, to remotely attest the integrity of a system, to protect code through obfuscation techniques and to build trusted overlay of virtual nodes.

2.3.1 Checking Kernel Integrity

To discover attacks against the kernel, the thesis proposes a solution that exploits VMI to monitor some kernel memory regions to detect illegal modifications made by an attacker trying to insert and execute arbitrary instructions, e.g. to modify the code of a system call. This poses the problem of the semantic gap [47] between the point of view of the guest OS in the Mon-VM, defined in term of processes, files, network connections, and the view that the VMM offers to the I-VM, defined in

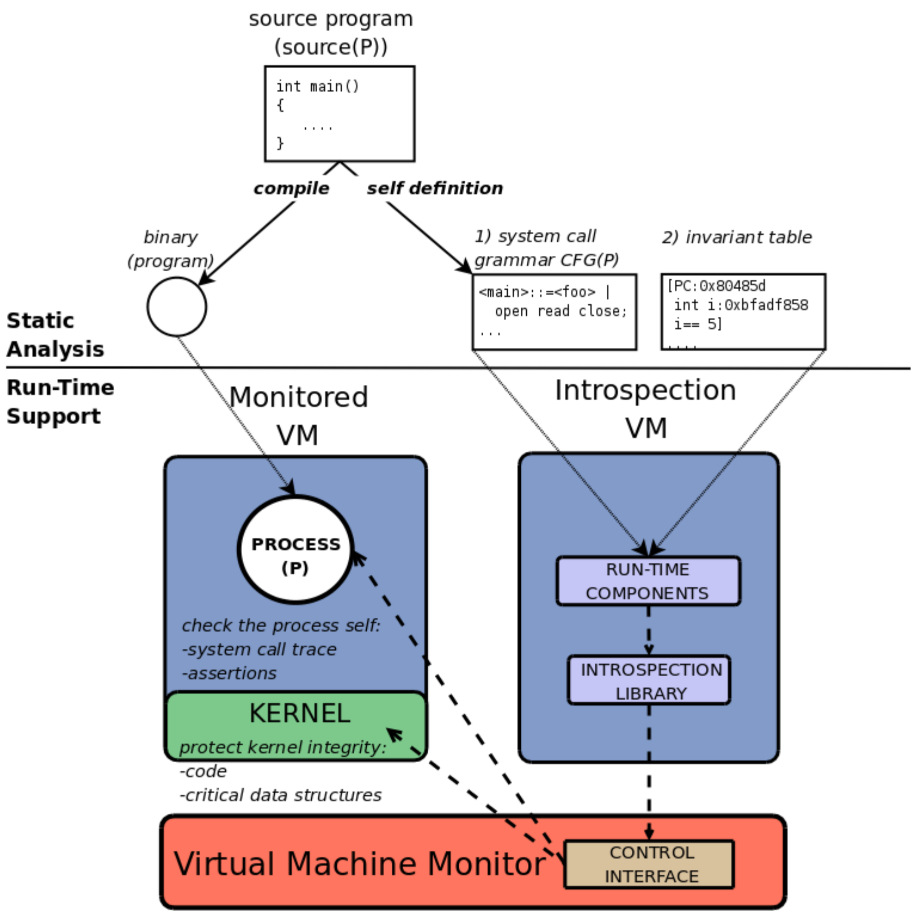


Figure 2.8: Overall PsycoTrace Architecture

terms of physical memory pages, CPU registers. To solve this problem, VMI needs to provide a high-level view of the state of the Mon-VM starting from the raw data accessed inside its memory. This is fundamental because the control interface of the VMM offers only a low-level view of the resources of the VMs, i.e. a view defined in terms of cells of memory, processor's registers and disk blocks, whereas a high-level view makes it possible to check kernel integrity by applying standard host intrusion detection techniques, defined in terms of OS resources, such as files or processes.

The transparency enabled by virtualization corresponds to the feasibility of checking the current state of the software on a Mon-VM both at the kernel and at the user-level without forcing the users to install any additional software. Two approaches to achieve transparency that we advocate in the thesis are:

- exploit processors with virtualization extensions, so that the VMM can trap the execution of the Mon-VM each time it executes some critical instructions, such as system calls;
- injecting a context-agent from the I-VM into the memory of the Mon-VM

to transparently obtain high-level information about the internal state of the kernel.

VMI-based context-agent injection is a mechanism to inject and protect a context-agent into a running VM using VMI without the cooperation of the monitored VM. This addresses the problem of obtaining reliable high-level information about the internal operation of the VM while having confidence that the context-agent has not been compromised.

2.3.1.1 Context-Agent

From the security point of view, context-agent injection has several benefits because the agent can provide high-level information about the health of the OS in the Mon-VM to enrich the view of the I-VM and also act on its commands, for example to remove malicious code, or install additional software, such as anti-virus updates. Moreover, the agent can neutralize a malware by undoing the malicious modifications to the kernel code or data-structures. A context-agent inside the Mon-VM may help PsycoTrace to bridge the semantic gap, because it can provide to the I-VM high-level information about the running system, such as the list of running processes, open files and network connections, logged users, running kernel modules. For example, the agent can send semantically rich notifications to the I-VM about event of interests, such as the creation of new tasks or processes, or the loading of new device drivers or kernel modules.

Nevertheless, the use of a context-agent presents two problems:

1. the complexity of installing and managing it;

2. the agent is vulnerable to attacks against the Mon-VM.

To solve both problems, we have designed and implemented a mechanism to inject and protect a context-agent into a running Mon-VM from an isolated I-VM, without the cooperation of the Mon-VM. A particular benefit of agent injection is that it enables the I-VM to make both persistent and non-persistent changes to the Mon-VM OS in a minimally invasive way. This allows the I-VM to trigger agents on demand to obtain information, without making permanent changes or installations on the system.

The ability of injecting arbitrary agents into running Mon-VMs also simplifies the management and deployment of patches in large virtual infrastructures. The scalability advantages are evident in cloud environments with several Mon-VMs where, instead of logging in all the machines and invoke the command to apply the updates, Mon-VMs can be constantly patched transparently from a single location. Moreover, while software to update the OS can be easily disabled by the users, the injection of patches from the I-VM guarantees that are always applied. Finally, agent injection can support compliance reporting and remediation, since the context-agent can also offer semantically rich and fine-grained information on the integrity of services and of data.

2.3.2 Remote Attestation of System Integrity

VMI enables the I-VM to verify the correct configuration and software integrity in the Mon-VM. The most popular mechanism to check the integrity of a tool or of a system configuration is that defined in the Trusted Computing framework and that computes a hash function on the sequence of memory positions that corresponds to the area that stores either the tools or a set of information about the configuration. However, this mechanism neglects loss of integrity due to run-time attacks. To take these attacks into account, we have to measure not only the integrity of the executables stored on files but also that of the software running in memory. The latter has to be measured at a rate that depends upon the security level that is required. This poses new problems because, first of all, the well-known execution environment initialized at boot time, and that provided a safe environment for the measurement, cannot be reproduced without rebooting the system. Second, since the applications data structures are continuously updated at run-time, their integrity cannot be checked through hash values.

The thesis shows how the framework underlying PsycoTrace can be generalized to cover the remote attestation of the integrity of a system. To do this, we consider an overlay, i.e. a virtual network, that offers some services. We assumes that node integrity is a precondition for joining the overlay without putting at risk both the security of the overlay and of the services it offers. This results in an architecture that attests the integrity of a node when it joins the overlay and it continuously monitors its integrity as long as it belongs to the overlay. The main goals of this architecture are:

1. measurements with a better detection capability than hash-based ones;
2. continuous measurement as long as a node belongs to the overlay;
3. a transparent attestation;
4. strong separation of the measurement system from applications;
5. avoid the introduction of privileged nodes;
6. minimization of the attestation overhead.

To satisfy these goals, first of all the architecture distinguishes the start-up attestation of the integrity of a node from the continuous monitoring of such integrity. The start-up attestation determines whether the node can join an overlay, whereas the monitoring aims to detect both attacks against P, which now runs the overlay application, and malicious updates to the node configuration.

The start-up attestation and the continuous monitoring apply distinct measurements according to the overlay security policy. To define and implement these measurements, we apply PsycoTrace. The start-up attestation evaluates a set of PsycoTrace assertions that generalize the hashing ones that a TPM applies to check software integrity. Continuous monitoring measurements compare P self against the actual behavior. Since the kernel integrity is initially attested, attacks that inject malicious code can be detected by monitoring the system calls that P issues and their parameters.

2.3.3 Code Obfuscation

This thesis introduces a new approach for code obfuscation that stems from virtualization and PsycoTrace run-time architecture. This approach exploits the decomposition of the program into system blocks, i.e. the fragments of code in-between two consecutive system calls. The Mon-VM only stores these blocks but they are encrypted and the Mon-VM does not store any information about their order. Instead, the control-flow logic of the program is represented by a system block graph that is safely stored only inside the I-VM, along with the keys to decrypt the system blocks. At run-time, the VMM traps the execution of a system call on the Mon-VM, freezes the Mon-VM and it transfer control to the I-VM that determines the next system block to be executed, decrypts this block, encrypts the previous one and modifies the program counter of the Mon-VM to point to the next system block.

2.3.4 Trusted Overlay of Virtual Networks

Virtualization can increase robustness not only by enabling more complete and severe controls on a VM, but also by replacing a physical node through a network of VMs and by partitioning the software node among these VMs to minimize the software each VM runs. These considerations result in the definition of a methodology adopts a highly parallel approach to share in a secure way an ICT infrastructure. To this purpose, it introduces several overlays, dynamically mapped onto the physical ICT infrastructure, where each physical node runs a VMM to multiplex the node's physical resources and strongly confine each VM. To simplify the configuration of the overlay, we introduce several VM templates each related to a specific applicative or system functionality. As an example, the execution of user applications is delegated to specialized Application VMs that only run the smallest number of software packages and libraries to support the corresponding applications. We introduce distinct Application VMs according to the privileges of the user and to the application's trust level, from high trust level applications, which access critical information, to insecure ones with low trust level, such as browsers. Other VM templates are introduced to:

1. manage shared resources among Application VMs of the same overlay or of distinct one,
2. control information flowing among VMs
3. map the overlays onto the physical infrastructure

To harden each VM by removing unneeded functionalities, the OS of each template can be carefully selected and configured according to its functionalities. The appropriate combination of OS and applications for each VM minimizes the overall complexity and increases the overall security [164], while preserving any software investment.

The number of overlays that share an infrastructure depends upon the number of user communities. Informally, a community consists of applications and of services to support these applications that can be handled in a uniform way because they have the same security and reliability requirements. While communities do not prevent cooperation and information exchange among user, the consistency and security checks that are applied within a community differ from those that are applied when crossing the community border. To define communities in a more formal way, we pair both any user and any application with a level that define, respectively, the security and the trust levels. The user security level defines the information the user can access while the user and application levels jointly define a further level that is coupled with the VM executing the application on behalf of the user. This level is a synthetic evaluation of:

- the actions that the application can execute;
- the resources that the VM can accesses;
- the reliability of service that the VM requires.

A community includes users and applications such that all the VMs that support the users and the applications have the same level. Distinct communities are implemented through distinct overlays. In this way, the VMs in an overlay can be homogeneously managed because they have similar requirements. As an example, data they exchange may be protected through the same mechanisms or they require the same reliability level. The homogeneous handling of the VMs in an overlay strongly simplifies the management of the overlays and their mapping onto the infrastructure physical nodes. While an overlay strongly resembles a VPN, an important difference lies in the granularity of the computation because we are interested in minimizing the complexity of the services a VM implements. As an example, some VMs may be introduced just to attest the integrity of, or to apply security checks to, other VMs. Furthermore, overlays are dynamically mapped onto physical networks.

Chapter 3

Related Works

This chapter reviews some of the main works on the topics considered by the thesis.

3.1 Sense of Self

[228, 116] firstly described a model that defines the self of a process in terms of a set of short sequences of system calls. Any sequence that does not belong to the statistical-based set is a signal of an intrusion. This approach was later exploited in [249], which compares four methods for characterizing normal behavior and detecting intrusions based on system calls in privileged processes. [41, 140] propose a solution that pairs a program with a specification of its intended behavior, i.e. the program policy. The proposed specification language is based upon predicate logic and regular expressions. A similar approach is discussed in [213], which exploits a language for capturing patterns of normal or abnormal behaviors of processes in terms of sequences of system calls and their arguments.

[212] proposes an approach to detect anomalous program behaviors through a finite-state automaton that learns a program behavior, expressed as sequences of system calls, and it does not require access to the program source code. [253, 151] propose data mining-based approaches to generate rules from system call sequences. Association rules and frequent episodes algorithms are used to compute the consistent patterns from audit data. [76] extends previous research on system call anomaly detection by incorporating dynamic window sizes and exploiting two methods: the first one is an entropy modeling method that determines the optimal single window size for the data, whereas the second method is a probability modeling method that takes into account context dependent window sizes.

Several authors [142, 234, 45] also propose a notion of anomaly detection that also considers constraints on system call parameters. [214] further explores the work described in [212] by discussing an approach where constraints on system call parameters are based on data-flow relations, i.e through policies able to specify that an argument of a system call is a function of arguments or return values of previous calls. An example is a policy that requires that the file descriptor argument for a `read()` system call is the value returned by a previous `open()` system call. Since source code is often unavailable, and static analysis of binary is rather complex on certain platforms, it would be important if advantages of the previous anomaly detection based models can be achieved without a static analysis of the source code or the binary.

[89, 90] introduce a new model of system call behavior, called an **execution graph**, which is a gray-box model that accepts only sequences of system calls that are consistent with the control-flow graph of the program, and it is maximal given a set of training data. This is the first model that does not require a static analysis of the program source or binary, and conforms to the control-flow graph of the program. [193] introduces an approach to system call monitoring based on **authenticated system calls**, i.e. system calls augmented with extra arguments that specify the policy for that call and a cryptographic message authentication code that guarantees the integrity of both the policy and the arguments. The kernel uses this extra information to verify the system call.

[243, 245] define a static analysis of the application source code that returns a specification of the expected application behavior. Intrusions are signaled by system call traces that are not coherent with the transition system that models the application. The paper introduces the **callgraph model**, built by analyzing the control-flow of the program. This model is then extended to the **abstract stack model** to take into account **impossible paths**, i.e. paths in the model that cannot be taken by the program. Finally, it considers the **digraph model** to simplify the implementation of the framework.

[99] proposes an approach to detect malicious system calls through a static analysis of the binary program that builds a model representing any remote call stream that the process may generate. As the process executes remotely, the local agent operates on the model incrementally, ensuring that any call received does not violate the model. Each model is defined in terms of finite-state machines. The control-flow graphs generated from the binary code are used to construct either a non-deterministic finite-state automaton or a push-down automaton to mirror the execution control-flow of the executable.

The **Dick Model** [100] includes a stack to record function call return locations, by using precalls and postcalls, and null system calls to eliminate impossible path and to simulate stack operations. **VtPath** [78] is an anomaly detection method that utilizes return address extracted from the call stack. It generates the abstract execution path between two execution points in the program and decides whether this path is valid according to what has been learned on the normal runs of the program. Moreover, since Pushdown automaton model are rather inefficient because of non-determinism, [79] explores the *VPStatic* model, a variant of the VtPath model, which extracts context information about stack activity of the monitored program to define a deterministic model. [98] further explores these idea by proposing a static data-flow analysis that associates a program's data-flow with specific calling contexts that use the data. Then, this analysis is exploited to differentiate system call arguments flowing from distinct call sites in the program.

[218] proves that for any system-call sequence model, under the same (static or dynamic) program analysis technique, there always exists a more precise control-flow sequence based model. [106] presents a new abstraction of program behavior referred to as an **Inlined Automaton Model** that is as accurate as a pushdown automaton model since it does not suffer from false positives, in the absence of recursion, and as efficient, in terms of run-time overhead, as a non-deterministic finite automata. The authors present a static analysis algorithm to build a control-flow and context-sensitive model of a program that allows for efficient on-line validation. [2, 1] describe a technique, based on the enforcement of **Control-Flow Integrity** (CFI), whose security policy, derived by a static binary analysis, dictates that software execution must follow a path of a control-flow graph. CFI enforcement is based on a combination of lightweight static verification and machine-code rewriting that instruments software with run-time checks. The run-time checks dynamically ensure that control-flow does not violate a given control-flow graph.

State-based control-flow integrity (SBCFI) [185] is an approach to dynamically monitor operating system kernel integrity based upon a property called state-based. Violations of SBCFI indicate a persistent, unauthorized modification of the kernels control-flow graph. The approach consists of of two steps: (i) validate kernel text, including static control-flow transfers, by keeping a copy or hash of the code; (ii) validate dynamic control-flow transfers. The latter requires the monitor to consider the dynamic state of the kernel, i.e. the heap, stack, and registers, to determine potential branch targets. The current implementation monitors function pointers inside the kernel. The monitor traverses the heap starting at a set of roots, which are global variables, and then it locates each function pointer that might be invoked in the future. It then verifies that these pointers target valid code, according to the control-flow graph. The algorithm identifies the roots, and then it builds the type graph and then uses this graph to generate the traversal code for the monitor and code to locate all function pointers reachable from global variables. Finally, function pointers are validated by checking whether the target of a pointer is consistent with the kernels control-flow graph. Four approximated candidates are introduced for determining consistency: (i) valid code region; (ii) valid functions; (iii) valid function type; (iv) valid points-to set.

Paid [148, 149, 251, 147] is a compiler-based intrusion detection system that derives an accurate system call model from the application source code. It derives a deterministic finite-state automaton model that captures system call sites, their ordering and partial control-flow information. Moreover, Paid exploits run-time information to minimize the degree of non-determinism of the system call graph returned by the static analysis, and it also computes a set of constraints on the arguments of sensitive system calls. A kernel run-time verifier compares the system call pattern of a process against the statically derived model.

[156] presents a mechanism for profiling the behavior space of an application by analyzing all function calls issued by the process, including regular functions and library calls, as well as system calls. Behavior is derived from aspects of both control and data-flow. The implemented system, called **Lugrind**, extracts the program behavior dynamically from the execution of the program binary without instrumenting the source code. The model is also based on a feature set that includes a mixture of parent functions and previous sibling functions and exploits the notion of Smart Error Virtualization, a self-healing technique that involves learning appropriate function return values at run-time.

[165] presents a novel approach to the analysis of system calls that uses a composition of dynamic analysis and learning techniques to characterize anomalous system call invocations in terms of both the invocation context and the parameters of the system calls. This model can also detect data modification attacks, which cannot be detected using only system call sequence analysis. The **SwitchBlade** [82] system integrates system call interception, in normal mode, and dynamic taint analysis, when checking violation of the system call model, and exploits randomization of the model to make code injections arbitrarily difficult. Moreover, the dynamic taint analysis integrates data-flow-based learning to update too strict models. The **guarded model** [200] is a generalization of previous models that offers no false alarms, a very low monitoring overhead, and is automatically generated. It detects mimicry attacks by combining control-flow and data flow analysis, and can also tackle non-control-data flow attacks. The model is built automatically by combining control-flow and data flow analysis using tools for automatic generation and propagation of invariants.

[27] proposes an extension to the Java security models by introducing **history-based policies**, which are expressed through finite state automata. History-based policies and mechanisms are alternative to stack inspection, and they depend on the whole execution of the program. A static analysis optimizes the run-time enforcement of policies and it exploits the call-graph construction and model-checking to predict those policies that will always be obeyed. At first, the static analysis extracts the control-flow graph, which is transformed into a history expression, then it model-checks the history expression against the usage policies. To specify, analyze and enforce safe usage of resources, the static analysis applies a model-checking tool [28], which runs in polynomial time in the size of the history expression extracted from the analyzed program. [161, 162] exploit abstract interpretation to derive a control-flow analysis that approximates the inter-procedural control-flow of function calls and returns by computing, for each expression, an abstract control stack. Control-flow analysis provides extra information about the points where a function returns at no additional cost by enabling the creation of more precise call graphs.

[67] presents an automated mechanism for generating robust signatures for kernel data-structures. This means that any attempt to evade the signature by modifying the structure contents will cause the OS to consider the object invalid. Through dynamic analysis, the target data structure are profiled to determine commonly used

fields, and then those fields are fuzzed to determine which are essential for the correct operations of the OS: these fields form the basis of a signature for the data structure. **ClearView** [182] is a system for automatically patching errors in software. To this end, it (i) observes normal executions to learn invariants that characterize the application's normal behavior; (ii) monitor the application's execution to detect failures; (iii) identifies violations of learned invariants occurring in a failed execution; (iv) generates candidate repair patches that enforce selected invariants by changing the state or the flow of control to make the invariant true; (v) observes the continued execution of patched applications to select the most successful patch.

Mimicry Attacks

[243, 244] firstly introduce the idea of **mimicry attacks**, i.e. any trace of system calls that does not trigger an IDS alarm and yet contains a malicious sequence of system calls. The author develops a tool that takes as input an attack sequence of system calls and outputs an entire valid sequence of system calls accepted by the IDS but where the system calls that do not belong to the attack are "nullified", i.e. the play the role of "semantic no-ops", and are present only to ensure that the IDS does not detect the attack. As an example, a way for transforming a system call in a no-op is to invoke it with an invalid argument, such as invoking `open()` with a non-existent pathname. When the system call fails, no action is taken, yet the IDS assumes that this system call was executed. Since most of the IDSes ignores the return value of system calls, this enables an attacker to nullify the effect of a system call while fooling the IDS into thinking that the system call succeeded.

[143] presents a novel technique to evade an IDS and to facilitate the task of an attacker to exploit a mimicry attack. Given a legitimate sequence of system calls, this technique allows the attacker to execute each system call in the correct execution context by obtaining and relinquishing the control of the application's execution flow through manipulation of code pointers. The author discusses a static analysis tool for Intel x86 binaries that uses symbolic execution to automatically identify instructions that can be used to redirect control-flow and to compute the necessary updates of the process environment.

To make mimicry attacks more difficult, [239] introduces the notion of **behavioral distance** to evaluate the extent to which processes, potentially running different programs and executing on different platforms, behave similarly in response to a common input. Inspired by evolutionary distance, the paper presents an algorithm to calculate behavioral distance and an algorithm to train the model to learn the behavioral distance automatically and shows that this approach holds promise for better intrusion detection with moderate overhead. [101] discusses a model to automate the discovery of mimicry attacks, by starting with two models: a program model of the application's system call behavior and a model of security-critical OS state. Given unsafe OS state configurations that describe the goals of an attack, the model finds system call sequences that are valid execution according to the program model but that nevertheless produce unsafe configurations. Moreover, a model checker attempts to prove that the attack effect will never hold in the program model. By finding counter-examples that cause the proof to fail, the model can find undetected attacks, such as system call sequences and arguments that are accepted as valid execution and induce the malicious attack effect upon the OS.

[257] proposes a mechanism, based on **waypoints**, to provide trustworthy control-flow information for anomaly monitoring and to detect global mimicry attacks. Waypoints are marks along the normal execution path that a process must follow to successfully access OS services. Waypoints actively log trustworthy context information as the program executes, allowing an IDS to both monitor control-flow and restrict system call permissions to conform to the legitimate needs of application functions. They can also catch return into-libc by guarding the return addresses.

To prevent mimicry attacks, [232] proposes an enhancement to the IDS by exploiting specifications to abstract the system call arguments and process credentials. The specification takes into account what objects in the system can be sensitive to potential attacks, and highlights the occurrence of unsafe operation. The model does not use the actual values of arguments and privileges, as this could result into a higher false positive rate, but rather it abstracts these values by categorizing them into distinct classes that are defined by a user-supplied category specification. To this end, an appropriate category specification should take into account the potential security impact of system call operations on system's objects and resources, such as files or directories.

[36] presents a novel defensive technique against mimicry attack, based on a "obfuscator" kernel module, which works in transparent way and with low overhead, that interacts with a host IDS and whose main scope is to randomize the sequences of system calls produced by an application to make them unpredictable by any attacker. The same approach is further extended in [37] where, by exploiting the Inter-procedural Control-Flow Graph of a protected binary, the authors propose a strategy where a static analysis techniques localizes critical regions of a program, which are code fragments that could be used to implement an automatic mimicry attack. Once these regions have been recognized, their code is instrumented to monitor the integrity of dangerous pointers during the execution of these regions, and any unauthorized modification will be undone by restoring at once the legal values.

[177] describes an alternative approach for building mimicry attacks that make these attacks a more immediate and realistic threat. These attacks, called **persistent interposition attacks**, are not as powerful as traditional mimicry attacks because an attacker cannot exploit them to obtain a root shell, but they enable cyber-criminals to achieve their goals, such as stealing credit-cards or hijacking and impersonating servers. Persistent interposition attacks are stealthier than mimicry attacks and are not IDS-specific and they can evade a large class of system-call-monitoring IDS, the **I/O-data-oblivious** ones. These IDSes have perfect knowledge of the values of all system call arguments as well as their relationships, with

the exception of data buffer arguments to `read()` and `write()` and thus they can be attacked by injecting code that interposes on I/O operations and modifying the data read or written by the victim but leaving the control-flow and other system-call arguments unmodified.

Data-Flow Integrity

[256] presents a solution called **Leapfrog** which retrofits binary executables with mandatory data-flow control, which enables a patched application to perform fine-grained data-flow control. Leapfrog exploits a technique that tracks sensitive data flows only at a small set of program locations, where each location uses the program's internal state and pre-computed conditions to predict the path taken by the data flows and the next location they will reach. To prevent attackers from exploiting buffer overflows and format string vulnerabilities to write data to unintended locations, [43] presents a technique that enforces **data-flow integrity**. This technique statically computes a data-flow graph and it instruments the program to ensure that at run-time the actual flow of data corresponds to the data-flow graph. This technique uses reaching definitions analysis to compute the data-flow graph and, for each value read by an instruction, it computes the set of instructions that may have produced the value. To enforce data-flow integrity at run-time, the implementation instruments the program to compute the definitions that actually reach each use by maintaining a table that identifies the last instruction that has written into each memory position. The program is instrumented to update this table before every write and to prevent the attacker from tampering with it. Also reads instruction are instrumented to check if the identifier of the instruction that wrote the value being read is an element of the set returned by the static analysis. In the same direction of the previous work, [31] presents an approach for enhancing the accuracy of host-based intrusion detection models by capturing data-flow information. This approach learns temporal properties involving the arguments of different system calls, thus capturing the flow of security-sensitive data through the program. Further works based on data-flow integrity are discussed in [84, 153, 258]

System Call Interposition

Janus [102] is a secure environment for untrusted helper applications, exemplified by browser plugins, that restricts the access of an untrusted program to the OS by using the OS process tracing facility. When the application attempts to invoke a system call, the framework dispatches this information to relevant policy modules. Each module reports its opinion on whether the system call should be executed, and any necessary action is taken by the framework. Each module contains a list of system calls that it will examine and filter and it may pair each system call with a function to validate the arguments before it is executed by the OS. Then, the function may use this information to update the local state, and then it may suggest allowing the system call, denying it, or make no comment on the attempted system call.

[87] presents techniques for developing Generic Software Wrappers, i.e. protected and non-bypassable kernel-resident software extensions to increase security without modifying the Commercial Off-the-Shelf component (COTS) source. It exploits a high-level Wrapper Definition Language, which enables wrappers to easily refer to collections of system calls and simplify the implementation of meaningful security functionality without requiring a complete knowledge of low-level kernel details. The strategy for providing protected and efficient enhancements for COTS systems is to implement a Wrapper Support Subsystem as a kernel module, to permit dynamic installation, and to track running processes and evaluate activation criteria at appropriate times to activate new wrapper instances for the processes.

[125] presents an implementation of a system-call interposition infrastructure at the user level, which offers similar level of security and comparable level of capabilities as kernel-based implementations of system call extensions. This implies that normal users can develop and deploy their own extensions; moreover, damage due to errors in the extension code is limited, and does not affect the security of the entire system. As a result, the infrastructure can be used to develop extensions that implement a variety of security-related tasks, such as custom auditing and logging, fine-grained access control, intrusion detection and confinement.

Program shepherding [138] is a method for monitoring at run-time control-flow transfers to enforce a policy that provides three techniques as building blocks for security policies. First, shepherding can restrict execution privileges on the basis of code origins by ensuring that malicious code masquerading as data is never executed. Second, shepherding can restrict control transfers based on instruction class, source, and target. As an example, shepherding can forbid execution of shared library code except through declared entry points, and can ensure that a return instruction only targets the instruction after a call. Finally, shepherding guarantees that sandboxing checks placed around any type of program operation will never be bypassed.

[91] discusses some of the problems and pitfalls of system call interposition, such as incorrectly replicating OS semantics, overlooking indirect paths to resources, race conditions, incorrectly subsetting a complex interface, and side effects of denying system calls. Then, it shows some practical solutions to these problems, and defines some general principles to avoid the pitfalls. **Ostia** [94] is a sandboxing that relies on a delegating architecture to overcome several limitations of sandboxing systems. Rather than introducing a sandboxed application that directly interacts with the kernel to access sensitive resources, the architecture delegates the responsibilities of obtaining those resources to an agent controlling the sandbox. This agent accesses resources on behalf of the sandboxed program according to a user-specified security policy.

e-NeXSh [132] is a security approach that exploits kernel and LIBC support to efficiently defend systems against process-subversion attacks. It monitors all LIBC function and system-call invocations, and validates them against process-specific information that strictly describes the acceptable program's behavior: any deviation from this behavior is considered malicious. The prototype is implemented as a set of modifications to the Linux kernel and a user-space shared library. The technique is transparent, requiring no modifications to existing libraries or applications.

3.2 Virtualization for Security

Virtual machine introspection (VMI) is first proposed in [95] together with Livewire, a prototype VMI IDS that monitor VMs through introspection. **Hyperspector** [141] is a monitoring environment to detect intrusions in a distributed system that separates the IDS from the system it monitors by running each IDS on a dedicated VM. Moreover, an independent virtual network connects all the IDS VMs. [150] presents a virtualization-based architecture to protect IDSes that exploits the confinement provided by a VMM to separate the IDSes from the monitored OS. It also provides a learning mode to build a database of sequences of invoked system calls.

ReVirt [69] is a logging system for VMs that supports *recovery*, *checkpoint* and *roll-back*. These techniques support *virtual-machine replay* because ReVirt can re-execute a system, encapsulated in a VM, instruction-by-instruction for recovering purposes. This concept is extended by **IntroVirt** [129], a virtualization-based system that detects intrusions by executing vulnerability-specific predicates.

[131] discusses a technique to debug Xen VM kernels through `gdb`. **Paladin** [23] is a framework that exploits virtualization to detect and contain rootkit attacks. It defines the notion of *protected zones*, which are guarded and protected from illegal access. These zones are partitioned into *Memory Protected Zone* (MPZ) and *File Protected Zone* (FPZ). The memory image of the kernel and the various jump tables are a part of the MPZ, which is set to non-writable so that any attempt to write into it will trigger an alarm. The FPZ includes the system files to be protected from being modified. The VMM intercepts system calls from the guest OS and forwards them to an application VMApp. Upon receiving a system call event, the VMApp process consults the policies specified and determines if the given system call violates any access control policies.

Manitou [155] is a system implemented within a VMM that ensures that a VM only executes authorized code by computing the hash of each page before executing the code it includes. Manitou sets the executable bit of the page only if the hash belongs to a list of authorized hashes. **XENKimono** [191] detects violations of the kernel security policy, by checking the kernel from a distinct VM through virtual machine introspection. XENKimono proposes two distinct strategies: (i) integrity checking to detect illegal changes to kernel code and jump-tables, e.g. system call table, IDT, page-fault handler; (ii) cross-view comparison to detect malicious modifications to critical kernel objects. Moreover, it monitors critical processes, detects suspicious activities and applies a white-list based detection, such as a list of applications that can have root access, a list of network ports that the applications can bind to and a list of kernel modules that can be loaded into the kernel.

Xenprobes [190] is a framework to probe several Xen guest kernels simultaneously and that allows developers to implement their probe handlers in user-space for kernel debugging purposes. **VMwatcher** [127] is an "out-of-the-box" approach that overcomes the semantic gap challenge. A new technique called guest *view casting* is developed to systematically reconstruct internal semantic views (e.g., files, processes, and kernel modules) of a VM from the outside in a non-intrusive manner. Specifically, the new technique casts semantic definitions of guest OS data structures and functions on VM states, so that the semantic view can be reconstructed. VMwatcher utilizes the symbol information exported by the kernel to apply guest view casting to identify and reconstruct critical guest data structures. VMwatcher enables developers to apply: (i) comparison-based stealthy malware detection, which compares a VM's semantic view obtained from both inside and outside to detect any discrepancy; (ii) out-of-the-box execution of off-the-shelf anti-malware software.

SecVisor [216] is a tiny hypervisor that ensures that only user-approved code can execute in kernel mode, by protecting the kernel against code injection attacks, such as kernel rootkits. SecVisor is effective even against an attacker who controls everything but the CPU, the memory controller, and system memory chips and can even defend against zero-day kernel exploits. SecVisor virtualizes the physical memory to set hardware protections over kernel memory, which are independent of any protections set by the kernel. SecVisor uses the IOMMU to protect approved code from DMA writes and it virtualizes the CPU's MMU and the IOMMU to ensure that SecVisor can intercept and check all modifications to MMU and IOMMU state.

XenAccess [38] is a monitoring library for OS running on Xen. XenAccess incorporates virtual memory introspection and virtual disk monitoring capabilities, so that monitor applications can safely and efficiently access the memory state and disk activity of a target OS. For example, these applications can retrieve the list of the running processes/modules, set watchpoints in privileged system directories to be notified each time those directories are updated. **Lares** [179] is a security tool that can actively control an application running in a guest VM by inserting hooks into the process execution flow. These hooks transfer control to another VM that checks the monitored application using introspection and security policies.

Lycosid [128] is a VMM-based hidden process detection and identification service. The key difference between Lycosid and previous VMM-based hidden process detectors is its use of implicitly information obtained by the monitored OS. Implicit information decouples Lycosid from the guest OS so that it can exploit at best its placement within a VMM. For example, Lycosid does not depend on the consistency of private guest OS data structures, so it is less vulnerable to guest-initiated evasion attacks. Similarly, Lycosid does not depend on guest OS implementation details, so it can be portable to several OSes. Lycosid exploits cross-view validation to detect maliciously hidden OS processes by comparing the lengths of the process lists obtained, respectively, at a low (trusted) and a high (untrusted) level. If the trusted list is longer than the untrusted one, Lycosid deduces that at least one process has been hidden. Lycosid applies *CPU inflation* technique, to allow a VMM to influence the run-time of specific processes by carefully patching their executable code. By forcing processes to run more frequently than they normally would, CPU inflation effectively increases the resolving power of Lycosid's identification techniques.

NICKLE [196] is a lightweight VMM-based system that transparently prevents unauthorized kernel code execution of unmodified commodity OSes. NICKLE is based on *memory shadowing*, wherein the trusted VMM maintains a shadow physical memory for a running VM and performs real-time kernel code authentication so that the shadow memory only stores authenticated kernel code. Further, NICKLE transparently routes guest kernel instruction fetches to the shadow memory. This guarantees that only the authenticated kernel code will be executed.

The **VIX tools** [114, 166] support a forensic analysis of a guest VM from a privileged VM. Using this approach, neither the virtual machines nor the virtual machine manager have to be modified. VIX consists of a library of common functions and of a suite of tools which mimic the behavior of common Unix command line utilities. The basic approach of these tools is to freeze the target virtual machine, acquire through read-only operations the data necessary to perform the requested function, and then resume the target VM. In this way, VIX can ensure that the state of the VM does not change during the data acquisition process, because it is not modified while the VM's execution is suspended.

Overshadow [49] presents an application with a normal view of its resources, but the OS with an encrypted view. This allows the OS to implement the complex task of managing resources, without allowing it to read or modify them. Overshadow exploits *multi-shadowing* which leverages the extra level of indirection offered by memory virtualization in a VMM. A VMM maintains a one-to one mapping from guest physical addresses into actual machine addresses, whereas multi-shadowing replaces this mapping with a one-to-many, context-dependent mapping, offering multiple views of guest memory. Overshadow leverages this mechanism to present an application with a clear-text view of its pages, and the OS with an encrypted view. Encryption-based protection allows resources to remain accessible to the OS, yet secure, so that the OS can manage them without compromising application privacy or integrity.

KernelGuard [195] is a solution that blocks dynamic data kernel rootkit attacks by monitoring kernel memory access using VMM policies. KernelGuard preemptively detects changes to monitored kernel data states and enables fine-grained inspection of memory accesses on dynamic kernel data. For each kernel data structure to be protected, a policy describes how the VMM should identify the data structure in a raw view of memory as well as the characteristics of an attack against that data structure. In addition, the policy describes the pointers within the kernel's memory that point to the data structure so that they can be tracked and protected as well. At run-time, the VMM locates the data structure in memory and intercepts all writes to its address in order to validate them and ensure they do not violate the policy. In addition, the specified pointers are also monitored in order to ensure that KernelGuard can monitor the data structure in real time even if the kernel loads it in memory, so that dynamic data structures are monitored as well.

Wizard [226] is a Xen-based kernel monitor. In contrast to virtual machine introspection, Wizard trusts no guest OS data, but its semantic understanding can identify kernel-level attacks that alter the kernel's execution behavior. Wizard monitors the interactions of a guest OS with the hypervisor and does not peer into memory states that may have been constructed by an attacker. Wizard verifies that the actual execution of each kernel service handler, as viewed by a hypervisor, does not differ from the execution of an unmodified, benign handler. Wizard detects operations of guest VMs that are visible outside the VM, such as system calls generated by user-level applications and VM calls or hypercalls generated by the guest kernel. Applications executing in the untrusted VM request kernel services by executing a software interrupt. Xen intercepts this interrupt, notifies the security software that a kernel handler will be executing, and passes the interrupt to the untrusted guest kernel. Xen also passes these events to Wizard's privileged VM component, and applications within the privileged VM use the event information to improve the security of a guest VM. When a guest VM generates an event, a handler inside Xen intercepts the system call or VM call and then writes the event's information into a shared memory region. Wizard logs both the system and VM calls together with their parameter values and the associated interrupt handler, the value of the CR3 register, and the entry and exit of interrupt handler execution. To verify the correlation between kernel service requests and the subsequent hardware accesses generated by the kernel handlers, Wizard requires a characterization of this correlation in during benign execution. To this end, during a training period, Wizard records the system call requests generated by all running applications and the subsequent VM calls produced by the kernel.

[186] presents a formal discussion of the development of VMI-based security applications by identifying those challenges that these applications should overcome and by defining a formal model for describing VMI techniques. **SADE** [50] is a stealthy deployment and execution mechanism to automatically inject agents into guest VMs. It also protect the integrity of in-guest agents at run-time and effectively hides the execution of agent code.

HookSafe [248] is a hypervisor-based lightweight system to protect several kernel hooks from being hijacked by kernel rootkits. HookSafe overcomes critical challenges of the protection granularity gap by introducing a thin hook indirection layer. Since any kernel hook, once initialized, is frequently read-accessed, but rarely write-accessed, HookSafe relocates those kernel hooks to a dedicated page-aligned memory space and then exploits hook indirection to regulate accesses to them through hardware-based page-level protection. [51] adopts virtualization to monitor and protect the systems running in a cloud from a centralized security VM. The proposed solution does not assume any a-priori semantic knowledge of the guest OS or any trust assumptions into the state of the VM.

Virtual Machine Detection and Attack

Virtual-machine based rootkit (VMBR) [137] is a type of malicious software that gains qualitatively more control over a system, with respect to standard malware. This malware installs a VMM underneath an existing OS and hosts the original OS into a VM. To insert itself beneath an existing system, a VMBR must manipulate the system boot sequence to ensure that it is loaded before the OS and applications. Then, the VMBR boots the target OS using the VMM. As a result, the target OS runs normally, but the VMBR sits silently beneath it. To install a VMBR on a computer, an attacker must first gain access to the system with sufficient privileges to modify the system boot sequence. After the attacker gains root privileges, she must install the VMBR's state on persistent storage. The most convenient storage for VMBR state is the disk. The paper explores the design and implementation of a VMBR, called **SubVirt**, and shows that the best way to detect a VMBR is to run at a layer that the VMBR cannot control. Detectors that run below the VMBR can see its state because their view of the system does not invoke the VMBR's virtualization layer. Such detection software can read physical memory or disk and look for signatures or anomalies that indicate the presence of a VMBR, such as a modified boot sequence. Other low-level techniques, such as secure boot, can ensure the integrity of the boot sequence and prevent a VMBR from gaining control before the target OS.

GuardHype [42] is a hypervisor with a focus on security and VMBR prevention that controls how the user deploys virtualization, allowing the execution of legitimate third-party hypervisors but disallowing VMBRs. GuardHype mediates the access of third-party hypervisors to the hardware virtualization extensions, effectively acting as a hypervisor for hypervisors. Another option relies on GuardHype to provide a standardized virtualization interface to which hosted hypervisors attach themselves to access the hypervisor layer. Malware usually include code to check for the presence of VMs, because these checks are straightforward, as the Intel IA-32 instruction set contains a number of instructions that are unvirtualizable [197]. Based on these instructions, a variety of detection techniques have been implemented, as an example the “Scooby Doo - VMware Fingerprint Suite” [139]. [92] surveys the wide range of dissimilarities between real and virtualized platforms, and the usage of timing benchmarks to detect the presence of VMMs.

[85, 86] investigate the problem of the remote detection of VMMs and devises **fuzzy benchmarking** to successfully detect the presence or absence of a VMM on a remote system. Fuzzy benchmarking works by measuring the execution time of particular code sequences on the remote system, by developing a fuzzy benchmarking program whose execution differs from the perspective of an external verifier when a target host is virtual machine. It exploits the timing dependency exception to the equivalence property of a VMM to detect the presence of a VMM without relying on implementation details or software artifacts. [189] describes a method for determining the presence of VM emulation in a non-privileged operating envi-

ronment. It use the Local Descriptor Table as a signature for virtualization. [192] surveys alternative strategies to detect system emulators.

Blue Pill [198] is a rootkit based on x86 virtualization technology that targets Microsoft's Windows Vista. It exploits AMD64 SVM extensions to move the OS into a VM on-the-fly and no modifications to BIOS, boot sector or system files are necessary. If an exact replica of the hardware state is exported on the VM, i.e. it allows nested virtualization, the solution to discover these rootkits is to verify that `MSR EFER.SVME` is equal 1, since this must be set to 1 before executing any SVM instruction. If, instead, the VMM exports a generic hardware interface inside a VM, without SVM, Blue Pill cannot install itself without hardware virtualization support.

Vitriol [81] is a proof-of-concept VM rootkit for Mac OS X using Intel VT-x. [80, 81] describes known attacks against several VM emulators: VMWare, VirtualPC, Parallels, Bochs, Hydra, QEMU, Atlantis, Sandbox, VirtualBox, CWSandbox and Xen, and describes ways to defend against them. [176] proposes an automatic technique to generate red-pills for detecting if a program is executed through a CPU emulator. The proposed technique has been implemented in a prototype, used to discover new red-pills for detecting two IA-32 CPU emulators, i.e. QEMU and Bochs, involving hundreds of different opcodes.

3.3 Hardware-based Security and Remote Attestation

[260] examines the effectiveness of secure coprocessor-based IDS where the IDS is run on a coprocessor rather than on the host. This means that a compromise of the host does not affect the coprocessor, and self-protection of the IDS monitor is achieved. Moreover, since a coprocessor can read the memory of the host, a coprocessor IDS can verify the correctness of the host's state. The advantages of a coprocessor-based IDS are: (i) independence from the host OS; (ii) narrow interface; (iii) secure boot; (iv) trusted observer, i.e., any authenticated statements made by the secure coprocessor can be fully trusted. However, given its external nature, a coprocessor IDS cannot interpose the host's execution the way that a host IDS can.

Copilot [183] is a coprocessor-based kernel integrity monitor for commodity systems designed to detect malicious modifications to the host kernel. It is transparent and can be expected to operate correctly even when the host kernel is thoroughly compromised. As an example, Copilot has been able to successfully detect the presence of several rootkits. The Copilot monitor consists of two machines and a PCI add-in card: the first machine is the monitored one and it contains the Copilot monitor on its PCI add-in card; the second machine is the admin station, where an administrator can interact with the Copilot monitor.

The Co-Processing Intrusion Detection System (**CuPIDS**) [252] project aims at improving information system security through dedicating computational resources to system security tasks in a shared resource, multi-processor architecture. The project explores the improvements that a symmetric multi-processing system offers over the traditional uni-processor model of security. The proposed approach runs a protected application on one processor while a shadow process specific for that application runs on a different processor and monitors the application process' activity, and responds immediately if the application violates policy. A prototype supporting fine-grained protection of the real-world application resulted in less than a 15% slowdown while demonstrating CuPIDS' ability to quickly detect illegitimate behavior, raise an alarm, automatically repair the damage due to the fault or attack, allow the application to resume execution, and export a signature for the dangerous activity.

[66] proposes a transparent and external approach to malware analysis, motivated by the intuition that a transparent malware analyzer should not induce any side-effects that are unconditionally detectable by malware. The paper proposes **Ether**, which is based on an application of hardware virtualization extensions, and resides completely outside of the target OS environment. Thus, there are no in-guest software components vulnerable to detection, and there are no shortcomings that arise from incomplete or inaccurate system emulation. Ether does not induce any unconditionally detectable side-effects by completely residing outside of the target OS environment. As a result, malware cannot detect Ether. The results of the experiments show that Ether remains transparent and defeats the obfuscation tools that evade the existing approaches.

Speculative Parallel Check (**Speck**) [169] is a system that accelerates security checks on commodity hardware by executing them in parallel on multiple cores. It provides an infrastructure that allows sequential invocations of a particular security check to run in parallel without sacrificing the system's safety. Speck creates parallelism in two ways: (i) it decouples a security check from an application by executing the application in a speculative way while the security check executes in parallel on another core; (ii) it creates parallelism between sequential invocations of a security check by running later checks in parallel with earlier ones. Speck provides a process-level replay system to deterministically and efficiently synchronize a security check and the original process. Speck has been tested to parallelize three security checks: sensitive data analysis, on-access virus scanning, and taint propagation.

[9] firstly proposed an architecture to check the integrity of a computer system through a integrity chain, under the assumption that the hardware is valid. It describes the **AEGIS** architecture for initializing a computer system by validating integrity at each layer transition in the bootstrap process. A description of the Trusted Computing Group (TCG) Trusted Platform Module (TPM) can be found in [22, 180].

Property based attestation [199] is a strategy that describes an aspect of the behavior of the platform to be attested with respect to security-related requirements. As an example, a property may state that a platform has built-in measures to conform to the privacy laws, or that it strictly separates processes from each other, or that it has built-in functionalities to provide Multi-Level Security. [188] presents a protocol and architecture for property based attestation that resolves scalability, privacy and openness issues raised by straightforward binary attestation using TPM hardware. In fact, with property attestation, a verifier is securely assured of security properties of the platform's execution environment without receiving detailed configuration data. This enhances privacy and scalability because the verifier needs to be aware of its few required security properties rather than a huge number of acceptable configurations.

[46] proposes a concrete and efficient property-based attestation protocol within an abstract model for the main functionalities provided by TCG-compliant platforms. **Semantic remote attestation** [111, 112] adopts language-based virtual machines for remote attestation of dynamic program properties. This increases the flexibility for the challenger, because the integrity monitor can examine the current state of a system to detect semantic integrity violations. While this technique alone will not produce complete results as it does not attempt to characterize the entire system, it does offer a way to measure the integrity of portions of the target not suitable for measurement by hashing.

[201] discusses the design and implementation of **Integrity Measurement Architecture** (IMA), which is a secure integrity measurement system for Linux. This architecture enables a system to prove that the integrity of a program on a remote system is sufficient. IMA uses the TPM to detect subversion of the measurements system by comparing a hash value stored in the TPM against the one in the measurement system audit log. **Attestation-based Remote Policy Enforcement** [203] is an access control architecture that enables corporations to verify client integrity properties using a TPM, and to establish trust upon the capability of the client to enforce the policy before allowing the client to access the corporate Intranet.

[207] proposes a trusted computing architecture to enforce access control policies in p2p applications. The proposed architecture is based on an abstract layer of trusted hardware that may be implemented though emerging trusted computing technologies. A trusted reference monitor monitors and verifies the integrity and properties of running applications through the functions of trusted computing and that can enforce various policies on behalf of object owners. Also user-based control policies are supported.

[174] examines whether trusted computing can remedy the relevant security problems in PCs and it argues that, although trusted computing has some merits, neither it provides a complete remedy nor it is likely to prevail in the PC mass market. [144] describes an open and scalable architecture for trusted virtualization. **Pioneer** [217, 215] is a software-based platform addressing the problem of verifiable code execution on legacy computing hosts without relying on secure co-processors or CPU virtualization extensions. Pioneer is based on a challenge-response protocol between an external trusted entity (the dispatcher) and an untrusted computing platform.

Prima [122] is an extension of the Linux IMA system to measure information flow integrity that can be verified by remote parties. **vTPM** [29] is a full software implementation of the TPM specification with further functions to create and destroy virtual TPM instances. The software is integrated in the Xen hypervisor to make TPM functions available to virtual machines. **Semantic integrity** [184] is a measurement approach targeting the dynamic state of the software during execution and, therefore, providing fresh measurement results.

UCLinux [145] is a Linux Security Module that enables TPM-based usage controls enforcement. It provides the attestation support, sealing support and protection from administrative abuse required by a trustworthy usage control system, and it does so with existing hardware and limited changes to an existing OS. [229] presents an efficient and portable TPM emulator for Unix that enables not only the implementation of flexible and low-cost test-beds and simulators but, in addition, provides programmers of trusted systems with a powerful testing and debugging tool that can also be used for educational purposes.

[73] introduces a technique that allows a hypervisor to safely share a TPM among its guest OSes and allows them full use of the TPM in legacy-compliant or functionally equivalent form. It also allows guests to use the authenticated-operation facilities of the TPM to authenticate themselves and their hosting environment. Moreover, the implementation makes use of the hardware TPM wherever possible, which means that guests can enjoy the hardware key protection offered by a physical TPM.

[209] proposes improvements for software-based attestation protocols by using the time stamping functionality of a TPM so that the execution time of the fingerprint computation can be measured locally. This also allows to uniquely identify the platform that is being verified. This solution can be further strengthen with a trusted boot-loader, which can identify the processor specification of the untrusted platform and provide accurate timing information about the checksum function.

[72] describes three practical techniques for authenticating the code and other execution state of an OS using the services of the TPM and a hypervisor. These techniques are specialized OS images, authentication of OS images with persistent state and virtual machine policy attestation. The techniques trade off detailed reporting of the OS code and configuration with the manageability and comprehensibility of reported configurations.

[108] extends behavior-based attestation to a **model-driven remote attestation** to prove that a remote system is trusted as defined by TCG. The described model-driven remote attestation verifies two compliance requirements to prove the trustworthiness of a remote system, i.e. expected and enforced behavior's compliance. [235] presents a non invasive method that respects a node privacy, but it only guarantees the integrity of anti-virus tools.

Assayer [178] is an architecture that leverages hardware-based attestation to enable end-hosts to embed secure proofs of code identity in packets. Recently, [168] a collaboration of companies has defined a framework to secure cloud computing with a hardware root-of-trust, by creating resource pools within private clouds that share common physical characteristics and the same security policies.

3.4 Code Obfuscation

[55] proposes a taxonomy of obfuscating transformations. The key to successful control transformations is the resilience of opaque predicates and variables [56], where opaque predicates and variables are constructs whose values are known to the obfuscator, but are difficult for the deobfuscator to deduce. An opaque predicate is trivial if a deobfuscator can deduce it by static local analysis, and weak if a deobfuscator can deduce it by static global analysis. Transformations that obscure data abstractions include modifying inheritance relations and restructuring data arrays [54]. [246] describes a set of transformations that introduce aliases and further hinder the analysis by a systematic break-down of the program control-flow that transform high-level control transfers into indirect addressing through aliased pointers. By doing so, the basic control-flow analysis is transformed into a general alias analysis, and the data-flow analysis and control-flow analysis are made co-dependent.

[154] describes and evaluates techniques to increase the complexity of disassembling, i.e. junk insertion, thwarting linear sweep and recursive traversal [210]. [57] identifies three types of attacks by malicious hosts on the intellectual property contained in software and describes powerful obfuscations techniques to obscure the control and data structures. [65] examines some sophisticated protection technologies available today and methods that anti-malware reverse engineers use to defeat them. [6] studies the pros and cons of virtualization to make distinct copies of a piece of software and to make them more tamper-resistant.

[236] presents a framework for quantitative analysis of control-flow obfuscating transformations by showing that several existing control-flow obfuscation techniques can be expressed as a sequence of basic transformations on the control-flow graphs. [118] describes three novel control-flow obfuscation methods for protecting Java class files, namely basic block fission obfuscation, intersecting loop obfuscation and replacing goto obfuscation. [32] describes Skype's protection mechanism and it shows that almost all of its code is encrypted at run-time and, finally, it proposes some reverse engineering attempts against it.

3.5 Collaborative Virtual Environments

Terra [93] is a VM-based architecture for trusted computing that enables applications with distinct security requirements to run simultaneously on commodity hardware. The software stack in each VM can be tailored to meet the security requirements of its applications. **PlanetLab** [52] is a global overlay network that runs concurrently multiple services in *slices*, i.e. networks of VMs that include some amount of processing, memory, storage and network resources. PlanetLab exploits the concept of an open grid of machines where resources can be dynamically allocated and discovered. **Poly^2** [39] is a framework aimed at segregating applications and networks and at minimizing OSes. The proposed approach maps network services onto different systems and it isolates specific classes of network traffic. To this purpose, administrative and application-specific traffic are mapped onto distinct networks. Moreover, minimized OSes should only provide the services required by a specific network application.

[133] propose a codification of the interactions required to negotiate the creation of new execution environments by modeling dynamic virtual environments (DVEs) as first-class entities in a distributed environment where Grid service interfaces negotiate creation, monitor properties, and manage lifetime. It also shows how DVEs can be implemented in a variety of technologies, such as sandboxes, virtual machines, or simply Unix accounts by evaluating costs associated with each approach. DVEs provide a basis for both customization of a remote computer to meet user needs and also enforcement of resource usage and security policies. They can also simplify the administration of virtual organizations (VOs), by allowing new environments to be created automatically, subject to local and VO policy. To protect file systems and network services from untrusted grid applications, **SVGrid** [263] introduces distinct execution environments for the applications and the storage areas.

sHype [202, 204] is a security architecture that controls the sharing of resources among VMs according to formal security policies at the VMM-level. It provides boot and run-time guarantees and addresses OS security weaknesses by providing confinement opportunities. It also enables secure communication and sharing between workloads on the same platform and potentially across multiple platform and organizational domains. The secure hypervisor enables its users to run a trusted operating system securely alongside a distributed operating system on a single platform. This idea is further explored with **Shamon** [160], an approach to securing distributed computation based on a shared reference monitor that enforces mandatory access control (MAC) policies across a distributed set of machines. Shamon enables the reference monitors on these machines to achieve a set of monitor guarantees. The implementation is based on the Xen hypervisor with a trusted MAC virtual machine built on Linux 2.6 whose reference monitor design requires only 13 authorization checks, 5 of which apply to normal processing, while the others are for policy setup.

Trusted Virtual Domains (TVDs) [107] is an architecture that offloads computing services into execution environments that demonstrably meet a set of security requirements. A TVD is an abstract union including an *initiator* and at least one *responder*. During the process of joining, all the parties specify and confirm the set of mutual requirements and each party is assured of the identity and integrity of the computer system of the remote party. The enforcement of the attestation is delegated to virtual environments.

[124, 123] present an implementation of MAC for Linux network communications that restricts socket accesses to labeled IPSec security associations. The Linux Security Modules framework defines a reference monitor interface that enables security modules to enforce comprehensive MAC for Linux 2.6. Socket communications are restricted by network interfaces and IP addresses but they cannot control access to particular applications on remote machines or reliably associate request processing with the appropriate remote principals.

[105] examines how to link specific properties of a remote system, verified through TPM-based attestation, to secure tunnel endpoints to counter attacks where a compromised authenticated SSL endpoint replays the TPM-based attestation of another system. The proposes mechanism can be deployed in virtualized environments to create SSL endpoint certificates and instant revocation that scales Internet-wide. [254] considers VMs as sandboxes that simplify the deployment of collaborative environments over wide-area networks. Each VM sandbox can be seen as a virtual appliance available to several users, so that new nodes can easily join, and be integrated into, the virtual network.

The **Dynamic Virtual Clustering** (DVC) system [71] integrates the Xen virtual machine with the Moab scheduler to enable the creation of virtual clusters on a per-job basis. These virtual clusters can provide a unique software environment for a particular application, or a consistent software environment across multiple heterogeneous clusters. **OurGrid SWAN** [44] provides a sandboxing environment for the OurGrid free-to-join grid. By executing the task on a VM it creates security guarantees to the grid resource's owner that are especially important for a free-to-join grid model. Grid task execution are improved by creating a common storage area that is only visible by the current running tasks of a given site. This avoids multiple transfer of a given file across inter-site networking because, after the first transfer, the file can be accessed via the intra-site networking.

[8] describes the development of a small and specialized environment based on a Mini-OS running on Xen to support sensitive applications. This environment is isolated from other domains and has a very small TCB. Both these properties increase the trustworthiness of the application and of the supporting environment, which is not intended as a general purpose OS. To maintain a small TCB, this environment does not include several features, such as file system support and networking. The papers shows that a lightweight library OS offers a convenient and practical way of reducing the trusted computing base of applications by running security sensitive components in separate Xen domains.

Trusted Grid Architecture [157] is a framework to build a trustworthy grid architecture by combining Trusted Computing and virtualization technologies. The proposed approach allows a user to check that a selected provider is in a trusted state before accessing a submitted grid job. Both the previous architectures consist of a grid of nodes where clients require services, using some form of negotiation to locate a trustworthy provider. [40] introduces a secure virtual networking model and a framework for efficient and security-enhanced network virtualization. The key drivers of this framework design are the security and management objectives of virtualized data centers, which are meant to co-host IT infrastructures belonging to multiple departments of an organization or even multiple organizations. The proposed framework merges existing networking technologies (such as Ethernet encapsulation, VLAN tagging, and VPN) and security policy enforcement to concretely support the abstraction of TVD, which can be thought of as security-enhanced variants of virtualized network zones. Policies are specified and enforced at the intra-TVD level (e.g., membership requirements) and inter-TVD level (e.g., information flow control).

[35] describes the requirements and services to ensure the scalable management and deployment of appliances implemented as VM images. These requirements are important to achieve scalability as well as develop methods to manage trust and enable more images to be deployed on more platforms in a secure manner according to site policies. Finally, it describes methods of adapting VM images to produce appliances and contextualizing them on deployment. **VM-FIT** [194] is an architecture that applies virtualization to build fault and intrusion tolerant network-based services. The VM-FIT infrastructure intercepts the client/service interaction at the hypervisor level, below the guest OS that hosts a service, and distributes requests to a replica group. The hypervisor is fully isolated from the guest operating system and provides a trusted component that is not affected by malicious intrusions into guest operating system, middle-ware, or service. Furthermore, it supports the implementation of more efficient strategies for proactive recovery in order to cope with the undetectability of malicious intrusions.

[96] presents a secure and flexible Enterprise Rights Management system based on a refined version of the TVD security model to establish isolated execution environments spanning over virtual entities across separate physical resources. The proposed security approach results in a two-layered policy enforcement on documents: a TVD Policy ensuring isolation of the workflow from other tasks on the user platforms, and a role-based document-policy ensuring both confidentiality and integrity of document parts. The proposes architecture offers advanced features for secure workflows such as offline access to documents and transparent encryption of documents exchanged via USB, external storage or VPN communication between peer platforms.

PEV architecture [126] is a formal integrity model to manage the integrity of arbitrary aspects of a virtualized system. This architecture is based upon a model that generalizes the TPM's integrity management functions to cover not only software binaries, but also VMs, virtual devices, and a wide range of security policies. PEV supports the verification of security compliance and the enforcement of security policies. **SnowFlock** [146] is a Xen-based implementation of the VM fork abstraction. VM fork enables cloud users and programmers to instantiate several VMs in different hosts in sub-second time, with little run-time overhead. Therefore, it enables the simple implementation and deployment of services based on familiar programming patterns that rely on the ability to quickly instantiate stateful workers.

Protected Storage

S4 [231] is a self-securing storage server that transparently maintains an efficient object-versioning system for its clients. By doing so, it prevents intruders from undetectably tampering with or permanently deleting stored data by internally auditing all requests and keep old versions of data for a window of time, regardless of the commands received from potentially compromised host operating systems. S4 uses a log-structured object system for data versions and a novel journal-based structure for metadata versions. In addition to reducing space utilization, journal-based metadata simplifies background compaction and reorganization for blocks shared across many versions.

Storage-based intrusion detection [181] enables storage systems to discover data modifications proper of system intrusions. This enables storage systems to spot several common intruder actions, such as adding backdoors, inserting Trojan horses, and tampering with audit logs. An IDS embedded in a storage device watches system activity from a new viewpoint, which immediately exposes some common intruder actions. Running on separate hardware, this functionality remains in place even when client OSes or user accounts are compromised.

[227] presents a survey of techniques for securely storing data, including theoretical approaches, prototype systems, and existing systems currently available. [219] improves backtracking techniques [136] by logging additional parameters of the file system (such as the offsets where a read or write operation is performed) during normal operations and examining the logged information during the analysis phase. In addition, it uses data flow analysis, e.g. reaching definitions, within the processes related to the intrusion to prune unwanted paths from the dependency graph. This results in significant reduction of the search space and time as well as the number of false positives.

[25] presents two storage-based IDSes for block storage environments and it shows that the impact on storage system performance is negligible. It exploits two alternative approaches to intrusion detection: a real-time one, which works at block storage level and another one that does not operate in a real time manner and is applied at file system level. It then discusses how intrusion detection schemes can be deployed as an appliance loosely coupled with a SAN storage system. The major advantage of this approach is that it is fully transparent because it uses the space and time efficient point-in-time copy operation of SAN storage devices.

SVFS (Secure Virtual File System) [264] uses virtualization technology to store sensitive files in a VM that is dedicated to providing secure data storage, and run applications in one or more guest virtual machines. Accesses to sensitive files are filtered by SVFS and are subject to access control policies. Because these policies are enforced independently in an isolated VM, intruders cannot bypass file protection by compromising a guest VM. In addition, SVFS introduces a virtual remote procedure call to improve the performance of data exchange crossing VM boundaries.

[113] presents two systematic threat modeling processes to base protection for

storage systems: (i) the CIAA process; (ii) the data life-cycle model process. The CIAA process structures threats and vulnerabilities into classes of attacks to match existing protection techniques for confidentiality, integrity, availability, and authentication. The data life-cycle process focuses on the most important asset of a storage system, and it traces the data life-cycle within an environment to ensure it is fully protected at each stage.

[24] presents a storage based IDS which uses time and space efficient point-in time copy and performs file system integrity checks to detect intrusions. The storage system software is enhanced to keep track of modified blocks to speed up a file system scan. Furthermore, when an intrusion occurs a recent undamaged copy of the storage is used to recover the compromised data. It proposes that the storage controller keeps track of modified blocks and uses this information to minimize the number of files to be examined. Furthermore, the system can rollback the state of one or more logical disks to a previous point in time to recover the compromised data.

[134] discusses important security issues related to storage and presents a comprehensive survey of the security services offered by existing storage systems. [262] presents a data management solution which allows fast VM instantiation and efficient run-time execution to support VMs as execution environments in Grid computing, which is based on novel distributed file system virtualization techniques. The proposed solution provides on-demand cross-domain access to VM state for unmodified VM monitors and enables private file system channels for VM instantiation by secure tunneling and session-key based authentication. Moreover, it supports user-level and write-back disk caches, per-application caching policies and middleware-driven consistency models and it leverages application-specific meta-data associated with files to expedite data transfers.

[261] presents a storage-based IDS that makes use of advantages of VM and smart disk technologies. The VMM can defend the IDS itself from potential attacks while the smart disk technology provides IDS with a whole view of the file system of the monitored VM. The virtual disk maintains a sector-to-file mapping table (the file-aware block level storage) and it can detect the changes to file content on-line. By exploiting these features, normal file-level intrusion detection rules can be converted into sector-level ones in order to integrate intrusion detection functions within the virtual storage ones.

VOFS (View-Only File System) [34] relies on trusted computing primitives and virtualization technology to provide a great level of security than current systems. In VOFS, a secure VM on the client authenticates itself with a content provider and downloads sensitive data. Before allowing the user to view the data in his or her non-secure VM, the VOFS client disables non-essential device output. This prevents the user, or any malicious software, from printing, uploading, or stealing the sensitive content. When the user has terminated the operations on a sensitive file, VOFS will reset the machine to a previous state and resume normal device activity.

[247] proposes a mechanism to provide secure and efficient access to large-scale

outsourced data, by encrypting every data block with a different key to achieve flexible cryptography-based access control. It exploits key derivation methods so that the owner has to maintain only a few secrets. Moreover, over-encryption and/or lazy revocation prevent revoked users from getting access to updated data blocks. [259] proposes a cryptographic network file system based on a MAC tree construction that uses a universal-hash based stateful MAC that results in standard model security proof and in better performance than a Merkle hash tree. The implementation is based on coreFS, a user-level network file system. [250] proposes an image management system that controls access to cloud's image repository, tracks the provenance of images, and provides users and administrators with efficient image filters and scanners that detect and repair security violations.

Part II

Principles and Implementation

Chapter 4

Description of the Process Self

In the previous chapters, we have shown that PsycoTrace strategies to detect attacks against a process are built around the notion of process self. The most powerful of these strategies describes the self as a set of legal traces of system calls, where each system call may be coupled with an assertion on some variables value.

In the following, we describe the two main PsycoTrace static tools that analyze $SourceCode(P)$ to extract $Self(P)$:

1. the first tool implements *Grammar Generating Algorithm* (GGA), an algorithm that we have defined to build $CFG(P)$, the grammar that describes the traces of P;

2. the second tool is the *Assertion Generator*, which generates $IT(P)$, which includes a set of invariants $\{I(P,1),\dots,I(P,n)\}$, that hold at point i of the process P.

4.1 Abstract Syntax Tree in PsycoTrace

Definition ($AST(P)$)**.** *$AST(P)$ is the abstract syntax tree coupled with P.*

A preliminary step of the static analysis builds $AST(P)$, which is exploited both to generate $CFG(P)$ and to produce the set of invariants. For this reason, the static tools require a detailed representation of $AST(P)$ at a proper level of abstraction. Some GCC compilation options[1] return an internal GCC representation of $AST(P)$ that a data-flow analysis cannot easily exploit to generate invariants. For this reason, the static tools exploit an alternative representation to the $AST(P)$ exported by the GCC, namely the one exported by Icaria-Ponder [10]. Icaria-Ponder performs a static analysis on the program and supports forward and backward slicing and chop [117, 121]. We have modified both a subset of the library (Ponder) and

[1]Such as `-fdump-translation-unit` and `-fdump-rtl-*`.

of the front-end (Icaria) to generate a DOT representation of the $AST(P)$ that enables the PsycoTrace static tools to (i) easily parse the DOT representation of the $AST(P)$ tree using existing libraries; (ii) generate images from the $AST(P)$ DOT representation: to this end, we have exploited Graphviz [70].

The nodes of the $AST(P)$ are represented by a record including the following fields:

1. a unique numeric identifier, to identify the node in the tree;
2. a type, which can be:
   - `ROOT`: it represents the root of the AST;
   - `LIST`: it is a special node used to list all the children coupled with a node;
   - `IDENTIFIER`: it represents the name for variables and functions;
   - `LITERAL`: it represents the actual value of a variable or of a function parameter;
   - `STATEMENT`: it represents the distinct types of the language statements;
   - `EXPRESSION`: it represents any type of a language expression, such as assignments, functions, mathematical/logical expressions;
   - `DECLARATION`: it represents a variable declaration;
   - `DECLARATOR`: it represents a function declaration;
   - `INIT_DECLARATOR`: it represents a variable declaration and its initialization;
   - `SPECIFIER`: it represents the type of a variable or the return type of a function;
   - `QUALIFIER` and `STORAGE`: they represent modifiers for functions or variables, such as `STATIC`, `CONST` or `EXTERN`;
   - `EMPTY`: it is used to uniformly represent all the language constructs inside an AST in all the cases where a node may be empty. As an example, the `ELSE` branch of an if-statement may be empty.
3. value or label: as an example, the name of a statement or the value of a literal;
4. the name of the file that defines an instruction;
5. the line number where an instruction appears;
6. the source code corresponding to the instruction in the AST.

4.2 Grammar Generating Algorithm

PsycoTrace static tools generate $CFG(P)$ by applying *grammar generating algorithm* (GGA) while traversing $AST(P)$. Formally, $CFG(P)$ is a tuple $< T, F, S, R >$, where:

- T is a set of terminal symbols with one symbol for each distinct system call in $SourceCode(P)$;
- F is a set of non-terminal symbols, one for each function defined in $SourceCode(P)$; each symbol corresponds to a subset of T that may be empty.
- $S \in F$ is the starting symbol, which corresponds to the `main` function;
- R is the set of production rules $X \rightarrow \beta$, where X is a non-terminal symbol and β a sequence of terminal and not-terminal symbols that may be empty.

Since $CFG(P)$ represents the control-flow of P in terms of legal traces of system calls, terminal and non-terminal symbols of $CFG(P)$ depend upon the system calls that $SourceCode(P)$ invokes and the functions it defines. Any rule defined by GGA exploits the type of a node of $AST(P)$ to deduce the type of the statements and to determine the structure of the corresponding production rule. As an example, a conditional `IF` includes the expression to be evaluated (the guard) and two children nodes, which correspond, respectively, to the instructions to be executed when the expression evaluates to `TRUE` or to `FALSE`. Each type of node corresponds to a semantic construct that is used to build $CFG(P)$. Thus, if a node of $AST(P)$ represents a function, each statement of this function is coupled with a distinct production, where the name of every production is built by appending the name of the file where the function is defined with the name of the function. In a similar way, GGA uniquely identifies the various instances of statements such as `FOR` or `WHILE` by appending a numerical subscript for each instance, e.g. $\texttt{FOR}_1$, $\texttt{WHILE}_2$.

4.2.1 Grammar Generating Algorithm Rules

GGA analyzes $AST(P)$ and for each function `fun` defined in $SourceCode(P)$ it inserts into F a new non-terminal symbol `FUN` and a new rule $\texttt{R}_{new}$ into R, where `FUN` is the left-hand-side of $\texttt{R}_{new}$. To generate the right-hand side of the rule, GGA linearly scans the definition of `fun` in $SourceCode(P)$. Distinct production rules may be generated, according to the type of statement met by GGA in the body of `fun`. For each statement, GGA generates a new rule and adds a new symbol to the right-hand side of $\texttt{R}_{new}$. In this way, $CFG(P)$ represents the system calls that `fun` can invoke and the ordering among the invocations in the body in `fun`.

To generate the rule for a statement, GGA considers the following cases[2]:

[2]The subscripts, such as `X`, `Y`, represent a unique numeric identifier to distinguish the distinct productions.

- if `fun` contains a block $\texttt{block}_1$ of instructions without conditional statements or loops, then:

  1. GGA produces a new rule $<\texttt{EXPR}_X> \rightarrow$ `syscall`, where `syscall` represents the system calls (if any) inside an expression in $\texttt{block}_1$.
  2. the symbol $\texttt{<EXPR}_X\texttt{>}$ is added to the right-hand-side of $\texttt{R}_{new}$.

- if `fun` contains a statement $\texttt{if(cond) block}_1$:

  1. GGA generates a new rule $<\texttt{STIF}_X> \rightarrow <\texttt{B}> \mid \epsilon$, where B is a new non-terminal symbol that represents the left-hand-side of the new rules generated by recursively applying GGA on $\texttt{block}_1$;
  2. the symbol $\texttt{<STIF}_X\texttt{>}$ is added to the right-hand-side of $\texttt{R}_{new}$.

- if `fun` contains a statement $\texttt{if(cond) block}_1\texttt{ else block}_2$:

  1. GGA generates a new rule $<\texttt{IFEL}_X> \rightarrow <\texttt{STIF}_Y> \mid <\texttt{ELSE}_K>$, where $\texttt{<STIF}_Y\texttt{>}$ and $\texttt{<ELSE}_K\texttt{>}$ are new non-terminal symbols that represent the left-hand-side of the two new rules generated by recursively applying GGA on $\texttt{block}_1$ and $\texttt{block}_2$;
  2. the symbol $\texttt{<IFEL}_X\texttt{>}$ is added to the right-hand-side of $\texttt{R}_{new}$.

- if `fun` contains a statement $\texttt{if(cond) \{block}_1\texttt{\} else if(cond) \{block}_2\texttt{\}}$ $\dots$ $\texttt{else if(cond) \{block}_n\texttt{\}}$ statement or, equivalently, $\texttt{switch (expr) case(val}_1\texttt{):}$ $\texttt{block}_1 \dots \texttt{case(val}_n\texttt{) block}_n$:

  1. GGA generates a new rule $<\texttt{STIF}_X> \rightarrow <\texttt{ELSE}_1> \mid <\texttt{ELSE}_2> \mid \dots \mid <\texttt{ELSE}_n>$, where $\texttt{<ELSE}_1\texttt{>}$, $\texttt{<ELSE}_2\texttt{>}$, $\texttt{<ELSE}_n\texttt{>}$ are the new non-terminal symbols of the left-hand-side of n new rules generated by recursively applying GGA on $\texttt{<block}_1\texttt{>}$, $\dots$, $\texttt{<block}_n\texttt{>}$;
  2. the symbol $\texttt{<STIF}_X\texttt{>}$ is added to the right-hand-side of $\texttt{R}_{new}$.

The previous rule is valid only if every `CASE` branch of the `SWITCH` construct contains a `break` statement. Otherwise, GGA manages the fall through scenario by combining all the `CASE` branches together. As an example the following `SWITCH` statement:

```
switch(expr) {
   case 0:  ...  break;
   case 1:  ...
   case 2:  ...
   case 3:  ...  break;
}
```

generates the production:

`<SWITCH`$_X$`>` $\rightarrow$ `CASE`$_0$ `|` `CASE`$_1$ `CASE`$_2$ `CASE`$_3$ `|` `CASE`$_2$ `CASE`$_3$ `|` `CASE`$_3$`;`

- if `fun` contains a statement `while(cond)` {`block`$_1$} or `for(init; control; iteration) block`$_1$:
  1. GGA generates, respectively, the rules:
     - <`WHILE`$_X$> $\rightarrow$ <`WHRE`$_Y$> <`WHILE`$_X$> | ϵ
     - <`FORS`$_X$> $\rightarrow$ <`FORE`$_Y$> <`FORS`$_X$> | ϵ

     where `<WHRE`$_X$`>`, `<WHRE`$_X$`>` and `<FORE`$_Y$`>` are two new productions rules generated by recursively applying GGA on `block`$_1$ [3];
  2. productions `<FORS`$_X$`>` $\rightarrow$ ϵ and `<WHILE`$_X$`>` $\rightarrow$ ϵ are inserted to handle the loop exit;
  3. a symbol `<FORS`$_X$`>` or `<WHILE`$_X$`>` are added to the right-hand-side of `R`$_{new}$.

Table 4.1 shows some simple examples of grammars generated by GGA. As another example, Tab. 4.3 shows $CFG(P)$ generated from $SourceCode(P)$ listed in Tab. 4.2. Appendix A discusses an alternative strategy that we have exploited to build $CFG(P)$.

4.3 Assertion Generator

The *Assertion Generator* is the static tool to generate invariants. To this purpose, $AST(P)$ is represented in a structured way that contains the following information:

- symbolic names of the variables;
- type of the variables;
- symbolic names of the functions containing the variables of interest;
- values that variables may assume statically;
- type of statements of the considered language (C), such as assignments, conditional statements, loop;
- memory size of variables, such as arrays with known size.

[3]Since expressions in conditional statements (e.g. `cond` in the `while` rule) may contain some system call invocations, these system calls have to be added to the terminal symbols of the rule as well.

```
f(){
   open();
   read();
   g();
   close();
}

g(){
  getpid();
}
```
	$\langle$F$\rangle\rightarrow$ **open read** $\langle$G$\rangle$ **close**;
	$\langle$G$\rangle\rightarrow$ **getpid**;

```
f(){
   open();
   if(x)
      read();
}
```
	$\langle$F$\rangle\rightarrow$ **open** $\langle ST_1\rangle$;
	$\langle ST_1\rangle\rightarrow$ **read** $\mid \epsilon$;

```
f(){
   open();
   if(x)
      read();
   else
      close();
}
```
	$\langle$F$\rangle\rightarrow$ **open** $\langle IFEL_1\rangle$;
	$\langle IFEL_1\rangle\rightarrow \langle STIF_2\rangle \mid \langle ELSE_3\rangle$;
	$\langle STIF_2\rangle\rightarrow$ **read**;
	$\langle ELSE_3\rangle\rightarrow$ **close**;

```
f(){
   open();
   while(x)
      read();
}
```
	$\langle$F$\rangle\rightarrow$ **open** $\langle WHILE_1\rangle$;
	$\langle WHILE_1\rangle\rightarrow \langle WHRE_2\rangle \langle WHILE_3\rangle \mid \epsilon$;
	$\langle WHRE_2\rangle\rightarrow$ **read**;

```
f(){
   open();
   read();
   if(execl(...) != -1)
      f();
   close();
}
```
	$\langle$F$\rangle\rightarrow$ **open read** $\langle STIF_1\rangle$ **close**;
	$\langle STIF_1\rangle\rightarrow$ **execl** $\langle$F$\rangle \mid \epsilon$;

Table 4.1: Examples of The Grammar Generating Algorithm

As previously said, $AST(P)$ is extracted from $SourceCode(P)$ by applying Icaria-Ponder and it includes all the previous information. The DOT representation of $AST(P)$ is then analyzed by exploiting the GRAPPA library to detect the type of

```c
int main(int argc, char **argv) {
   char *filename;
   int fd;
   filename = argv[1];
   fd = fork();
   if(fork() == 0) {
      if(execl(filename, NULL) == -1) {
         printf("ERROR!\n");
      }
      else {
         strcat(filename, ".execl");
      }
   }
   else { ... }

   if(argc > 2) {
      filename = argv[2];
      strcat(filename, ".exec2");
   }

   execl(filename, NULL);
}
```

Table 4.2: Example of Source Code for P

nodes inside the tree. We have extended this library by defining some functions that, given a node, return, respectively, the depth, the father, the children and the siblings of the considered node.

The basic strategy of the Assertion Generator to generate invariants is to traverse $AST(P)$ and analyze the variables, functions and language statements to build the *invariant table* ($IT(P)$), which includes a set of invariants $\{I(P,1),\ldots,I(P,n)\}$, each associated with a program point i where P invokes a system call. To simplify the analysis, we assume that the following assumptions hold:

- *integer variables*: we restrict the analysis to files and socket descriptors so that we can express relations among these variables and the system calls;
- *string variables*: in case of arrays of char statically declared, functions to manipulate strings, such as `str(n)cpy` and `str(n)cat`, are treated like assignments. In the following, these kinds of functions will be denoted as `ATOMIC_FUN`;
- *struct members*: we restrict the analysis to integer or string type field.

4.3.1 Invariant Table

When analyzing $AST(P)$, the Assertion Generator builds the invariant table ($IT(P)$), which contains the following fields:

$\langle \text{MAIN}_P \rangle \rightarrow \langle \text{EXPR}_0 \rangle \ \langle \text{EXPR}_1 \rangle \ \langle \text{IFEL}_2 \rangle \ \langle \text{STIF}_{11} \rangle \ \langle \text{EXPR}_{14} \rangle;$

$\langle \text{EXPR}_0 \rangle \rightarrow$ /*empty*/;

$\langle \text{EXPR}_1 \rangle \rightarrow$ **fork**;

$\langle \text{IFEL}_2 \rangle \rightarrow \langle \text{STIF}_3 \rangle \mid \langle \text{ELSE}_4 \rangle;$

$\langle \text{ELSE}_4 \rangle \rightarrow$ **fork** $\langle \text{EXPR}_{10} \rangle;$

$\langle \text{STIF}_3 \rangle \rightarrow$ **fork** $\langle \text{IFEL}_5 \rangle;$

$\langle \text{IFEL}_5 \rangle \rightarrow \langle \text{STIF}_6 \rangle \mid \langle \text{ELSE}_7 \rangle;$

$\langle \text{ELSE}_7 \rangle \rightarrow$ **execl** $\langle \text{EXPR}_9 \rangle;$

$\langle \text{STIF}_6 \rangle \rightarrow$ **execl** $\langle \text{IFEL}_8 \rangle;$

$\langle \text{EXPR}_8 \rangle \rightarrow \langle \text{PRINTF}_p \rangle;$

$\langle \text{PRINTF}_p \rangle \rightarrow \ldots;$

$\langle \text{EXPR}_9 \rangle \rightarrow \langle \text{STRCAT}_{p_1} \rangle;$

$\langle \text{STRCAT}_{p_1} \rangle \rightarrow \ldots;$

$\langle \text{EXPR}_{10} \rangle \rightarrow \ldots;$

$\langle \text{STIF}_{11} \rangle \rightarrow$ /*empty*/ $\mid \langle \text{EXPR}_{12} \rangle \ \langle \text{EXPR}_{13} \rangle$

$\langle \text{EXPR}_{12} \rangle \rightarrow$ /*empty*/;

$\langle \text{EXPR}_{13} \rangle \rightarrow \langle \text{STRCAT}_{p_1} \rangle;$

$\langle \text{EXPR}_{14} \rangle \rightarrow$ **execl**;

Table 4.3: Context-Free Grammar for P

- `VAR_ID`: name of the variable in the source code. If `VAR_ID` appears on the right-hand-side of an assignment, we say that `VAR_ID` is an *assignment variable*, and we refer to its value as a *assignment value*. If `filename` is the name of a variable used as the first parameter of the `open()` system call, then we will say that `param`$_0$`@open = filename` is an invariant where `filename` is an assignment variable and the subscript 0 indicates the first parameter, 1 the second parameter, and so on;

- `FUN_FILE`: the name of the function or file that defines `VAR_ID`;

- `VALUES`: a set of values that `VAR_ID` can assume;

- `RUN-TIME`: if the variable value can only be known at run-time;

- `TYPE`: the type of `VAR_ID`. It also indicates whether a variable is local to a function or is static;

- `LENGTH`: dimension (if defined) coupled with `VAR_ID`;
- `LOC`: line number where `VAR_ID` appears;
- `REL`: relational operator of the invariant;
- `INVARIANTS`: it corresponds to `VAR_ID REL VALUES`;
- `RULE`: name of the rule where the current relation appears.

To produce $IT(P)$, the Assertion Generator implements a depth first visit of $AST(P)$, from left to right, so that it can firstly meet variable declarations and then statements or compound statements. Every time the Assertion Generator meets a declaration of a variable that belongs to the set of types of interest, it adds the variable to $IT(P)$, by initializing `VALUES` with an undefined value[4] (?). If the declaration is also an assignment (`INIT-DECLARATOR`) that can be statically determined, then `VALUES` is initialized with the corresponding value. Every time the Assertion Generator finds an `EXPR-ASSIGNMENT` statement or an `ATOMIC-FUN` that can be statically determined, it checks if the variable belongs to $IT(P)$, and it updates `VALUES` with the corresponding value, otherwise it evaluates the next statement. In case of conditional statements where the Assertion Generator cannot determine the value of a variable, $IT(P)$ is modified to include a union of all the possible values that can be assigned to the corresponding variable, and if they are statically determined or known at run-time. In this case, the Assertion Generator considers the control/data-flow path to determine the values that can be assigned to the variable.

As an example, in the following code:

```
1 int a;
2 if(test) {
3    a = 1;
4 }
5 else {
6    a = 2;
7    ref(a);
8 }
```

at the end of the evaluation of the `IF` branch (line 4), the set of values that the variable `a` may assume is $\{?, 1\}$, whereas at the end of the `ELSE` branch (line 8) the set is $\{1, 2\}$.

If the value of the variable involved in an `INIT-DECLARATOR` or `EXPR-ASSIGNMENT` is not statically computable, then the `RUN-TIME` column in $IT(P)$ is updated with the value `RT` to indicate that the value of this variable can only be known at run-time. By default, the value of the parameters of the `main` function are set to `RT`.

[4]We assume that an undefined variable means that its value has been initialized by the compiler to a default value according to the type of the variable, e.g. 0 for integer variables

Whenever a variable is referred, the largest set of values it can assume is returned by determining the set of reaching definitions. This set is then correlated to the call-site by specifying the referred variable, the name of the function (also for system calls), the grammar rule where the referred variable appears and the line in the code of the call-site. To this end, during the visit of the $AST(P)$, the Assertion Generator keeps track of the type of the variables, their scope (local or global), their dimension (if any) and the line of code where the variable appears. The first prototype determines the reaching definitions by visiting the $AST(P)$ and by considering the control/data-flow to determine the values that a variable can assume before a point of execution. The Assertion Generator checks the current statement `S` and:

- if `S` is not a conditional statement, then the Assertion Generator assumes that the set of values of the predecessors is the same as the last set of values in `VALUES`, since this models the correct control/data-flow, as previously described;
- if, instead, `S` is a conditional statement, firstly the Assertion Generator considers potential assignments in a branch and then it analyzes the stack of previous assignments by selecting the set of predecessor values of the statement. As an example, in the code previously described, the set of predecessors of `a` at line 7 is equal to $\{2\}$ since the assignment to `a` in the `ELSE` branch is the only point of execution that influences the current value of the variable at line 7. If `a` has not been assigned at line 6, then the set of predecessors of `a` would have been $\{?\}$, since the statement corresponding to the `IF` branch is not considered in the control flow that reaches the variable referred at line 7.

To generate the relations (`REL`) and the corresponding invariants (`INVARIANTS`) the Assertion Generator applies a contextual analysis during the visit of the $AST(P)$. In the following, for the sake of simplicity, we only focus on unary equality relations and on system call parameters. As an example, it is well known that several equality relations hold among file descriptors in a sequence of system calls such as `open()`, `read()`, `write()`, `close()` that involve the same file.

An Example. Let us consider the code in Tab. 4.4: the corresponding $IT(P)$ is shown in Tab. 4.5, and it contains two main pieces of information:

1. all the possible values that the variables declared in the program (`argc, argv, test, path1, path2, fd`) can assume;
2. the values of the program variables when they are referred in the code, as previously described. These values are used to generate invariants.

As far as concerns system calls parameters, we can isolate the following invariants:

```
 1  /*
 2     invariant generation example
 3   */
 4
 5  int main(int argc, char **argv) {
 6     char path1[10];
 7     char path2[10] = "B" ;
 8     int test, fd;
 9     if(atoi(argv[1]) == 0) {
10        test = 2;
11        strcpy(path2, "D");
12        strcpy(path1, "D");
13        fd = open("file_path", "RW");
14     }
15     if(atoi(argv[1]) > 5) {
16        test = 1;
17        strcpy(path1, "A");
18        printf("%s", path1);
19        write(fd, path1, sizeof(path1));
20     }
21     else {
22        test = 0;
23        execl(path2, "");
24        printf("%s", path1);
25    }
26  }
```

Table 4.4: Example of Source Code for P

- `13. PRM`$_0$`@open = {"file_path"}`
- `13. PRM`$_1$`@open = {"RW"}`
- `19. FD`$_0$`@write = {?,OPEN@13}`
- `19. path1`$_1$`@write = {"A"}`

In the first two cases, the $IT(P)$ suggests that the first and second parameter of the `open()` system call should be equal to, respectively, `"file_path"` and `"RW"`. On the contrary, in the last two cases, the $IT(P)$ indicates the first parameter of the system call `write()` is either undefined (`?`), i.e. it has the default value according to the type of the variable, or equal to the value returned by the `open()` system call issued at line 13: in this case, by applying introspection at one of the previous system call invocations (after `open()`), this value can be retrieved at run-time. On the other hand, the second value of the system call `write()` is statically known and is equal to `A`. In fact, the variable `path1` is assigned by the function `strcpy` at line 17. By computing the reaching definitions, the Assertion Generator can deduce the set of values that a referred variable can assume. In fact, at the beginning of the `IF` statement at line 15, the variable `path1` can assume the values {`?,D`}, but that is not longer true at `write()` call site, since the `strcpy` function overwrites the set of possible values with the single value `A`.

Rule	**INVARIANT**	**RT**	**LOC**
PRM$_{FUN}$@main	argc = {?}	RT	5
PRM$_{FUN}$@main	argv = {?}	RT	5
main_stringsSYS	path1 = {?}	-	6
STIF$_0$	path1 = {?,"D"}	-	12
STIF$_6$	path1 = {?,"D","A"}	-	17
STIF$_6$	path1 = {?,"D","A"}	-	18
ELSE$_7$	path1 = {?,"D","A"}	-	24
main_stringsSYS	path2 = {"B"}	-	7
STIF$_0$	path2 = {?,"B","D"}	-	11
ELSE$_7$	path2 = {?,"B","D"}	-	23
main_stringsSYS	test = {?}	-	8
STIF$_0$	test = {?,2}	-	10
STIF$_6$	test = {?,2,1}	-	16
ELSE$_7$	test = {2,1,0}	-	22
main_stringsSYS	fd = {?}	-	8
STIF$_0$	fd = {?,open}	RT	13
STIF$_6$	fd = {?,open}	RT	19
EXPR$_2$	path2$_0$@strcpy$_0$ = {"B"}	-	11
EXPR$_2$	PRM$_1$@strcpy$_0$ = {"D"}	-	11
EXPR$_3$	path1$_0$@strcpy$_1$ = {?}	-	12
EXPR$_3$	PRM$_1$@strcpy$_1$ = {"D"}	-	12
EXPR$_4$	PRM$_0$@open$_2$ = {"file_path"}	-	13
EXPR$_4$	PRM$_1$@open$_2$ = {"RW"}	-	13
EXPR$_8$	path1$_0$@strcpy$_4$ = {?,"D"}	-	17
EXPR$_8$	PRM$_1$@strcpy$_4$ = {"A"}	-	17
EXPR$_9$	PRM$_0$@printf$_5$ = {"%s"}	-	18
EXPR$_9$	path1$_1$@printf$_5$ = {"A"}	-	18
EXPR$_{10}$	fd$_0$@write$_5$ = {?,OPEN@13}	RT	19
EXPR$_{11}$	path1$_1$@write$_6$ = {"A"}	-	19
EXPR$_{12}$	path2$_0$@execl$_7$ = {"B","D"}	-	23
EXPR$_{13}$	PRM$_1$@execl$_7$ = {""}	-	23
EXPR$_{14}$	PRM$_0$@printf$_8$ = {"%s"}	-	24
EXPR$_{15}$	path1$_1$@printf$_8$ = {?,"D"}	-	24

Table 4.5: Invariant Table for P

It is worth noticing that the invariant `fd`$_0$`@write`$_5$ `= {?,OPEN@13}` on the `write()` parameter helps the discovery of a logic error inside the program. In fact, the value of `fd`$_0$ may be undefined when it is used. Therefore, as a side-effect, the $IT(P)$ can also be helpful in discovering common flaws, such as uninitialized variables.

As previously said, to increase the generality of the Assertion Generator, the $IT(P)$ is exploited to infer invariants involving any program variable. As an example, when the `open()` system call is issued, the Assertion Generator can also check the possible values of the variable `test`. Since the last useful assignment of this variable occurs at line 10, whereas the `open()` is invoked at line 13, the corresponding invariant is equal to:

$$\texttt{13.}\quad \texttt{test}_1\texttt{@open}_2 \texttt{ = 2}$$

Obviously, the invariant is coupled with the first system call where it can be evaluated, i.e. the `open()` system call. In the same way, let us consider the last `execl()` system call inside the `else` branch. In this case, the corresponding invariant is equal to:

$$\begin{array}{ll}\texttt{23.} & \texttt{path2}_0\texttt{@execl}_7 \texttt{ = } \{\texttt{"B","D"}\} \\ \texttt{23.} & \texttt{PRM}_1\texttt{@execl}_7 \texttt{ = } \{\texttt{""}\}\end{array}$$

Also in this case, the value that `path2` may assume at the execution of `execl()` is either `B` or `D`.

PsycoTrace can exploit run-time information to reduce the set of alternative values that a variable can assume. In other words, it can propagate the knowledge obtained at run-time to reduce the size of this set. Let us consider again the previous example. Statically, PsycoTrace can identify the following invariants:

$$\begin{array}{ll}\texttt{23.} & \texttt{path2}_0\texttt{@execl}_7 \texttt{ = } \{\texttt{"B","D"}\} \\ \texttt{23.} & \texttt{PRM}_1\texttt{@execl}_7 \texttt{ = } \{\texttt{""}\}\end{array}$$

By accessing run-time information through introspection, PsycoTrace run-time tools can further specialize the invariant by reducing the set of values that `path2` may assume. As an example, if the first `open()` at line 13 is executed, PsycoTrace can access the final value of `path2`, since this value is not changed anymore until the execution of the `execl()` system call at line 23. This means that the block inside the first `IF` is executed, and the value `D` has been assigned permanently to the variable `path2`. If PsycoTrace propagates this knowledge, the previous invariant can be tightened as follows:

$$\begin{array}{ll}\texttt{23.} & \texttt{path2}_0\texttt{@execl}_7 \texttt{ = } \{\texttt{"D"}\} \\ \texttt{23.} & \texttt{PRM}_1\texttt{@execl}_7 \texttt{ = } \{\texttt{""}\}\end{array}$$

4.3.2 Assignment Variables

In general, the values of most variables involved in any invariant is known at run-time only. The main problem when evaluating an invariant is due to the temporal relation and to the synchronous approach that PsycoTrace run-time tools exploit to access the state of the processes. In fact, PsycoTrace introspection mechanism can access memory locations and processor's registers only when a system call is issued. By exploiting the definition of System Block (SB) (discussed Chap. 7) as the program fragment in-between two consecutive system calls, then introspection may be applied only at the begin and end of a SB. If we consider invariants that involve assignment variables, we should correctly determine their assignment value. Let us consider two kinds of invariants:

- those involving variables whose assignment value is statically known: they are evaluated at the begin and end of any SB;
- those involving variables whose assignment value is only known at run-time; in general, they cannot be evaluated at the begin and end of any SB.

If we consider the following snippet of code:

```
...
char *filename;
L0: uid_t uid = getuid();
L1: filename = function(...);
L2: Log(...);
L3: execl(filename, NULL);
...
```

the invariant that is statically generated and that corresponds to the `execl()` system call is:

$$\mathtt{PRM}_0\mathtt{@execl=filename@L1}$$
$$\mathtt{PRM}_1\mathtt{@execl=NULL}$$

In this case, the value of `filename` is known only in-between the `getuid()` executed at `L0` and the `execl()` at `L3`. If we assume that `function()` may determine a value which is not related to values known before the `getuid()` system call, even if PsycoTrace knew the address of the `filename` variable, this invariant *cannot* be evaluated at run-time, because the evaluation can only be coupled with the execution of the `execl()` system call, where no information about the assignment value is known. This problem arises because PsycoTrace cannot access the state of P as soon as the assignment values, referred by some invariants, are modified, but *only when a system call is issued*. In other words, the analysis of the state of P is based upon system blocks rather than upon data-flow notions. This implies that some invariants that can be statically inferred, cannot be evaluated at run-time or can only be partially evaluated. In fact, there are some cases where PsycoTrace can access the assignment value of a variable, but this value may not be correct. Let us consider the following example :

```
...
char *filename;
L0: filename = function(...);
L1: Log(...);
L2: uid_t uid = getuid();
. . .
L3: execl(filename, NULL);
...
```

When the `execl()` system call is invoked at L3, PsycoTrace can evaluate the value of `filename` to check the correctness of the parameters of the system call. However, before the execution of the `getuid()` system call, which is the first point where PsycoTrace run-time tools can retrieve any variable value and check an invariant, the function `Log()` is invoked. If this function changes the value of `filename`, the evaluation of the invariant at `execl()` may use an inconsistent value, which can lead to a false negative. This problem is due to the fact that also functions may invoke system calls but the current prototype only analyzes functions defined in $SourceCode(P)$ and not those defined inside shared libraries. This problem can be solved by analyzing the object code rather than the source code, but this strongly increases the complexity of generating invariants.

With respect to assignment variables, there are four classes of invariants:

1. invariants that do not involve assignment variables, such as:

   15. $\texttt{test@open}_2\ \texttt{= 2}$

2. invariants involving assignment variables whose value is statically known;
3. invariants involving assignment variables whose value is only known at run-time and that can be determined inside a SB;
4. invariants involving assignment variables whose value is only known at run-time and that cannot be determined inside a SB.

To determine the value of a assignment variable that can be known at run-time only, two conditions must hold:

1. the assignment variable must not be declared or reassigned inside the SB;
2. no function is invoked in-between an assignment to the variable and the beginning of a SB.

Suppose k is the assignment variable involved in an invariant, i is the entry-point of a SB, o the exit point of a SB, and j the point of execution where k takes its final value. Notice that this point is only known at run-time. Then, the assignment value of k can be determined at the entry of SB if and only if:

1. k is assigned in j and $j < i$;
2. there is no statement of type `EXPR-ASSIGNMENT` or `ATOMIC-FUN` in-between i and o;
3. no function that may update k is invoked in-between i and o.

If the previous conditions are satisfied, then PsycoTrace can determine at i the value of k that the invariant uses at o. In other words, PsycoTrace can determine at the entry of SB the value of all the variables, such as k, that are involved in the evaluation of the invariant at the exit point of SB. Otherwise, PsycoTrace may adopt the solution proposed in Sect. 5.3.3.

Chapter 5

Run-Time Architecture

This chapter firstly describes the run-time tools that check and protect the kernel integrity and then those that check and protect the process self.

5.1 Run-Time Components

The implementation of the run-time tools is built around *Virtual Machine Introspection* (VMI) to apply introspection at the hardware/firmware level without introducing additional units. The run-time architecture consists of two virtual machines:

1. the monitored VM (Mon-VM), i.e. the VM executing P;
2. the introspection VM (I-VM), i.e. the VM monitoring P through virtual machine introspection.

The I-VM can access each component of the Mon-VM to inspect its running state both to check the process self of P and the kernel integrity. To implement these checks, the Mon-VM transfers control to the I-VM each time P invokes a system call. At this point, the I-VM checks that the current trace of P satisfies $CFG(P)$ and it evaluates the invariant $I(P, i)$ coupled with the point i reached by P. Periodically, the I-VM applies a set of integrity functions to assure the kernel integrity. Figure 5.1 shows the overall architecture.

The I-VM runs an *Assertion Checker* that evaluates invariants on the state of P and that accesses the variables in the memory of P and the CPU of the Mon-VM through an Introspection Library, which has a low-level access to each component on the Mon-VM. Every time P issues a system call, the Mon-VM transfers control to the I-VM, which:

(i) retrieves the system call number and the value of its parameters from the processor registers of the Mon-VM (e.g., `EAX`, `EBX`, `ECX`, `EDX`);

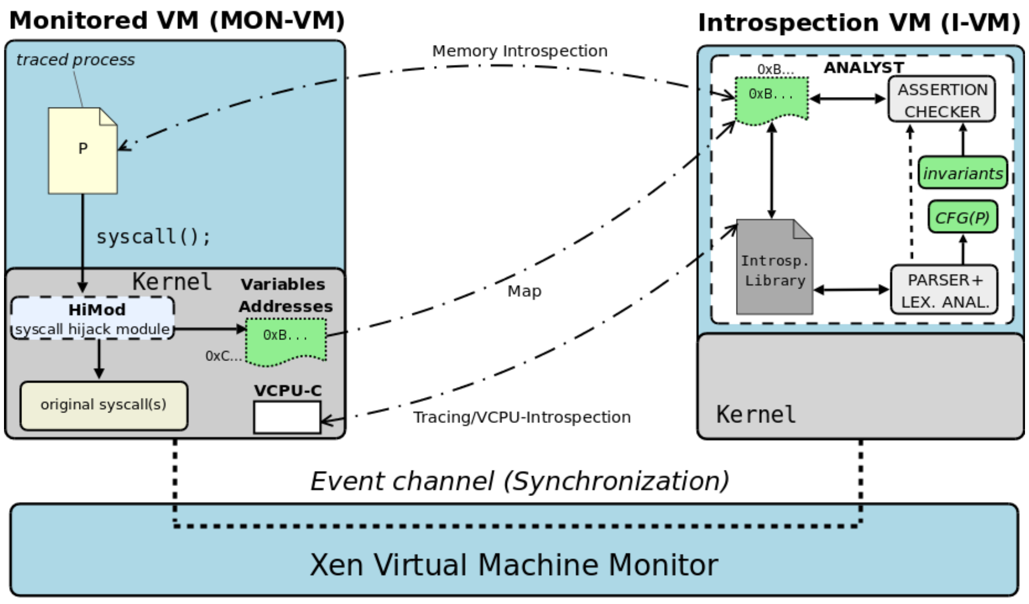


Figure 5.1: PsycoTrace Run-Time Architecture

(ii) determines the invariant coupled with the program counter (PC) corresponding to system call that P has issued;

(iii) retrieves the values of the variables that the invariant refers to;

(iv) evaluates the invariant and:

- kills P if the invariant is false, because this signals a successful attack against P;
- otherwise it resumes the execution of P by returning control to the Mon-VM.

To create the Mon-VM and I-VM, we have adopted Xen [26] mainly because of its high performance and complete integration with the Linux kernel.

5.1.1 Assumptions

The important assumptions underlying the adopted architecture are that:

(i) the VMM can be trusted, i.e. it belongs to the Trusted Computing Base (TCB);

(ii) introspection safely extends the TCB.

To justify these assumptions consider that, first of all, the VMM is more robust than commodity OSes because:

- it exports a simple interface to the higher levels, which is more difficult to subvert than, for example, the one of a kernel that implements hundreds of system calls;
- the small size of its code reduces the likelihood of a compromise and makes it possible to formally validate its correct implementation.

Secondly, the VMM has full visibility of the Mon-VM, because it can access every component of this VM. Notice also that the kernel of the Mon-VM does not belong to the TCB because the I-VM can check its integrity, as discussed in Sect. 5.2. In conclusion, since the VMM has full visibility of the VMs but it is strongly isolated from them, the complexity of compromising the VMM or of eluding the introspection monitoring capabilities of the Mon-VM is very high. Nonetheless, as discussed in Related Works, there are known threats against the VMM that also have to be considered.

5.1.2 Transparency

To be fully transparent, PsycoTrace should not require any modification of $SourceCode(P)$ or of the kernel of the Mon-VM. Currently, while $SourceCode(P)$ is not modified to build $CFG(P)$, the generation of invariants requires some updates to $SourceCode(P)$ to enable the I-VM to retrieve the addresses of local variables because they are dynamic and cannot be extracted at compile time. Hence, we update $SourceCode(P)$ to store their run-time addresses in a shared memory-mapped page through Xen grant-tables. A fully transparent strategy may be defined by statically computing variable addresses as the relative offset of local variables from the frame pointer and by accessing the frame pointer at run-time through VMI. Furthermore, PsycoTrace requires some modifications to the kernel of the Mon-VM to intercept each system call and trace P (see Sec. 5.3.1). There are at least three alternative implementations of a fully transparent system even with respect to the Mon-VM:

1. IDT-hijacking: when the Mon-VM kernel tries to install its own interrupt table, the VMM installs a different system call handler, which traces system calls and then jumps to the original system call handler. This handler can be installed when the Mon-VM kernel invokes the `set_trap_table()` hypercall to submit a table of trap handlers. This solution does not work with `syscall/sysexit` instructions, and it requires a para-virtualized Mon-VM kernel, i.e. modified to invoke hypercalls;

2. exploiting the hardware `NX` bit, for example to set to read-only the page storing the system call handler. Each time P invokes a system call, a trap is generated and Xen can trace the system call. Then, it sets to read-only the executable bit of the pages storing, respectively, the system call handler and the `ret_from_syscall()` procedure, so that the return of the call generates a

further trap. When handling this trap, PsycoTrace can set the executable bit of both pages to read only again, and so on;

3. in case of full virtualization, and with AMD processors with VT extensions, the VMM can be instrumented to invoke the `VMEXIT` AMD instruction, which is an instruction that traps the execution of the Mon-VM, each time P executes an `int $0x80` instruction, i.e. a software interrupt instruction to invoke a system call. This solution does not work with `syscall/sysexit` instructions, which are instructions to implement fast system call invocations.

The adoption of any of these solutions results in a monitoring that is both *fully transparent* to P and Mon-VM kernel and highly robust.

5.2 Kernel Integrity

As previously discussed, the most sophisticated attacks strives to occupy the lowest system level, i.e. the kernel-level, for at least two reasons:

(i) to achieve more complete control of the system;

(ii) to deceive the legitimate owner of the system, by hiding the traces of the compromise.

To detect these attacks, an IDS running on the I-VM may exploit the VMM direct access to the memory of each Mon-VM and apply VMI to analyze the state of the kernel hosted on a Mon-VM, so that integrity checks are applied at a lower level than the one a rootkit can gain. A possible solution runs on the I-VM a modified IDS so that it can work at the VMM level and use VMI to check intrusions. A further, distinct, approach applies VMI to evaluate a set of consistency checks defined according to the OS-level semantics. In this approach, an introspection library rebuilds the high-level view of the Mon-VM that the IDS requires. Both approaches are feasible, but the effort of the first one is very high because the IDS should be modified to monitor the Mon-VMs at the hardware level. On the other hand, the second approach requires a complex introspection library able to bridge the semantic gap and offer an OS view in terms of files, processes, virtual memory and that should also support the various versions of the OSes of interest. PsycoTrace defines a hybrid approach by introducing its own *Introspection Library* to rebuild the status of the processes and of critical kernel data-structures starting form the status of the memory and of the CPU of the Mon-VM. The Introspection Library enables the I-VM to build a high-level view of a Mon-VM state by mapping the raw data accessed in the Mon-VM memory into a high-level view in terms of OS data structures. To this purpose, the library needs to know the kernel hosted by the Mon-VM, the data structures it uses and the memory areas where they are allocated. In this way, the I-VM can consider global information about the OS of the Mon-VM such as the list of running processes, the list of the loaded modules or information associated with a PID, such as the list of open files/sockets. However, to support more complex OS concepts while bounding the complexity of the Introspection Library, PsycoTrace can inject a context-agent (see Sect. 5.2.3) in the Mon-VM's memory to retrieve further data it needs to complement the functionalities of the Introspection Library.

5.2.1 Introspection Library

A critical problem underlying the evaluation of assertions is related to the mechanism to access any memory region of P to retrieve the values of interest. As described in the following, this problem is rather complex and its solution strongly influences the overall performance. The I-VM implements this access through the Introspection Library because it enables the I-VM to traverse the page tables of P to translate a virtual address of a variable of P into a machine address, i.e. a physical address in Xen terminology. In this way, the I-VM can map any page of P into its address space to access the variables of P. The Introspection Library has been implemented and tested on 32-bit x86 architectures both with regular paging and Physical Address Extension (PAE), in the two cases of para-virtualized OS guest or full-virtualized VMs. The library implements two introspection functions, namely *Memory Introspection*, to access the memory of a Mon-VM both at the user and at the kernel level, and *VCPU-Context Introspection*, to retrieve the state of the Mon-VM virtual processor. These two functions are described in the following.

5.2.1.1 Memory Introspection

To implement user-space memory introspection, the library needs to access any physical memory location allocated to the Mon-VM that corresponds to a virtual address of P. To translate this virtual address, the Introspection Library directly accesses the page tables (PTs) of P and then follows the pointer to walk the paging levels to retrieve the pairing between a virtual address and a physical one. In the case of para-virtualization, the addresses in all the page levels and in the registers of a virtual context of a VM are machine addresses. For instance, the page directory address in the `cr3` register is a machine address. This implies that the Introspection Library has to map three pages to translate a virtual address into a machine address and it maps the corresponding page using the `xc_map_foreign_range()` function, as shown in Fig. 5.2.

Conversely, Xen manages static addresses as pseudo-physical addresses in a para-virtualized OS, such as those coupled with the kernel exported symbols. Hence, when Xen starts a VM, kernel static addresses are relocated, and the original addresses are managed as pseudo-physical ones. For this reason, the Introspection Library translates a pseudo-physical address PPA coupled with a kernel symbol, by applying the following four steps (see Fig.5.3):

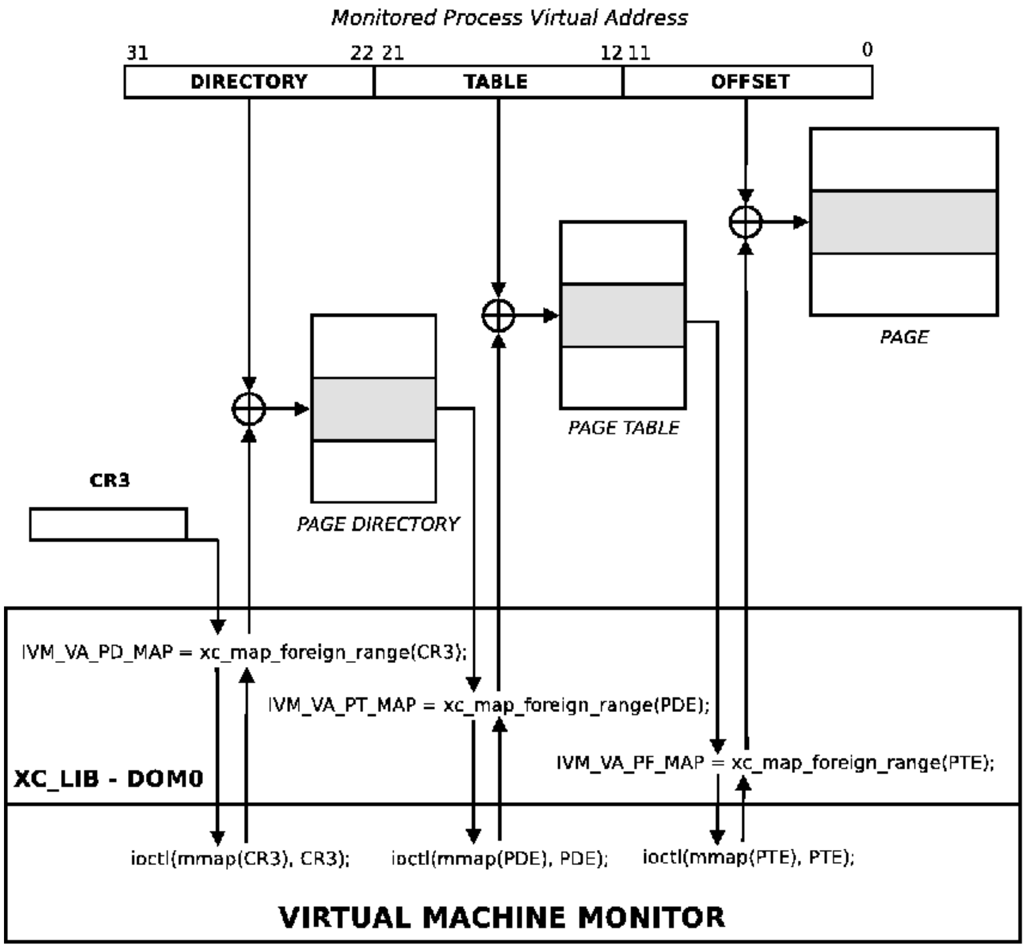


Figure 5.2: Introspection Library: User-Space Page Mapping

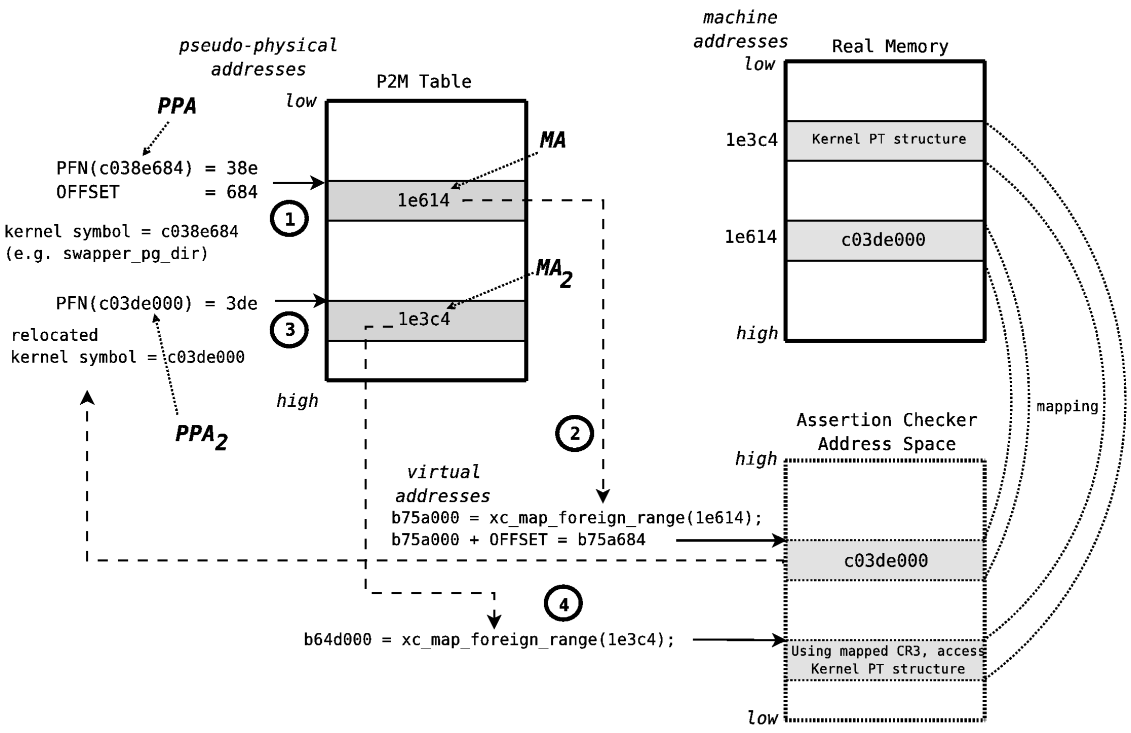
1. translate PPA into a machine address MA using the physical-to-machine (P2M) table. Note that MA does not reference the kernel symbol because it is relocated, i.e. Xen adds a further level of indirection to the kernel pseudo-physical addresses;

2. invoke Xen to map the page at the base address of MA, i.e. the page that includes MA, and retrieve from the resulting offset the relocated pseudo-physical address PPA_2 of the kernel symbol;

3. access the P2M table to translate PPA_2 into the corresponding machine address MA_2;

4. request Xen to map the page at the base address of MA_2 into the address space of the Assertion Checker process. This page stores the kernel data structure pointed to by the kernel symbol.

As soon as the Introspection Library has mapped the page that stores the pointer to the kernel page directory and referenced to by the `swapper_pg_dir` symbol, it can translate pseudo-physical addresses by accessing the kernel PTs as in the case of a process virtual address. In this case, the Introspection Library sequentially maps three pages instead of executing the previous four steps. Finally, when exploiting processor virtualization extensions, Xen applies the shadow PTs mechanism and both the page directory and PTs store pseudo-physical addresses. Each time a PT needs to be updated, Xen propagates the update to the real PT, which is known to the MMU. To this end, the Introspection Library exploits the Xen `page_array` structure, which records the pairing between pseudo-physical frame numbers and machines frame numbers.



pseudo-physical addresses
PPA
PFN(c038e684) = 38e
OFFSET = 684
kernel symbol = c038e684
(e.g. swapper_pg_dir)
PFN(c03de000) = 3de
relocated kernel symbol = c03de000
PPA2
P2M Table
low
high
1e614
1e3c4
MA
MA2
1
2
3
4
virtual addresses
b75a000 = xc_map_foreign_range(1e614);
b75a000 + OFFSET = b75a684
b64d000 = xc_map_foreign_range(1e3c4);
machine addresses
Real Memory
Kernel PT structure
c03de000
1e3c4
1e614
mapping
Assertion Checker Address Space
c03de000
Using mapped CR3, access Kernel PT structure

Figure 5.3: Introspection Library: Kernel Memory Access

5.2.1.2 VCPU-Context Introspection

To support the context switch between two VMs, Xen saves the values of the CPU registers in a *Virtual CPU-Context* coupled with each VM. When a VM $\mathtt{vm}_a$ is going to be scheduled, the current values of the registers are saved into the VCPU context of the running VM, while the values of the registers of $\mathtt{vm}_a$ are restored from the proper VCPU context. The VCPU-Context Introspection allows the I-VM to monitor, and modify, the content of any Mon-VM register. As an example, it enables the I-VM to retrieve the current system call number by reading the `EAX` register and the value of any parameter stored in or pointed to by a VCPU register or to monitor, and modify, the content of any Mon-VM register.

The VCPU-Context Introspection function exploits a Xen data structure, which is `vcpu_guest_context_t`, which contains the following fields:

- `unsigned long ctrlreg[8]`, the control registers for the virtual CPU. As an example, the control registers can be used to access the page directory through the `CR3` register;
- `struct cpu_user_regs user_regs`, the user registers, such as the `EIP` and all the registers used to save the parameters of a system call.

5.2.2 Integrity Checks

The I-VM monitors the kernel text section and that of the loaded modules to discover whether they have been modified, for example by a kernel-level rootkit. Since these memory regions are read-only, any attempt to modify them implies that an attacker is trying to insert and execute arbitrary instructions, usually to update the code of a system call.

The basic approach to retrieve a Mon-VM's kernel data-structure to check its integrity is:

1. the I-VM freezes the execution of the Mon-VM;
2. the I-VM maps into its address space the pages in the kernel of Mon-VM that store the data structures of interest[1];
3. by exploiting the definitions in the kernel header files, the Introspection Library in the I-VM casts the raw memory to the correct data-structure;
4. if the data-structure contains a pointer, the I-VM retrieves the corresponding virtual address and retrieves and maps the pointed data-structure, as in 3;
5. the I-VM applies consistency checks and returns to 3 if the data-structure is a list;
6. the I-VM resumes the execution of the Mon-VM.

Appendix B lists the code of a sample Introspection function to retrieve the list of running processes in Mon-VM that exploits this approach.

The first version of PsycoTrace's kernel integrity functions have been developed using the C language and exploits some security functions defined by OpenSSL. Each Mon-VM runs a Linux Debian distribution.

[1]This step exploits the `System.map` file to retrieve the kernel virtual address of the data structures.

The following sections discuss some sample introspection functions to exemplify some of the capabilities of PsycoTrace's kernel integrity functions.

5.2.2.1 Detecting Kernel Modifications

The I-VM invokes this introspection function to check the pages of the OS kernel that should never be modified and that store:

- the kernel code, from the address `_text` to `_etext`;
- `sys_call_table`, the system call dispatch table;
- `idt_table` table, the interrupt descriptor table.

The I-VM periodically computes the hash of each page and verifies that it has not been changed by comparing it against the original value.

5.2.2.2 Running Processes Checker

The I-VM retrieves a set of PIDs by rebuilding the list of the processes executed on a Mon-VM pointed by the `init_task` symbol. Then, it compares this set against the one returned by a context-agent injected into the Mon-VM. If the two sets of PIDs differ, then an attacker has replaced critical system binaries with trojaned versions to hide her presence. If the number of running processes is known, the list of allowed PIDs and the name of the processes can be fixed when the Mon-VM is booted and a hash of each element in the list is periodically checked.

5.2.2.3 Open Files Checker

This function is similar to the previous one, as it retrieves the list of running processes, and for each process it rebuilds:

- the list of files that the process has open through the `open()` system call;
- the list of memory mapped files, such as shared libraries (in Linux, `.so` files) that the Linux Loader loads into the process address space, or loaded at run-time through the `mmap()` system call.

Fig. 5.4 shows the kernel data-structures that the Introspection Library rebuilds from the raw data to retrieve the list of open files.

5.2.2.4 Loaded Modules Authenticator

This function retrieves the list of the modules loaded into the kernel, which is pointed by the `modules` symbol. Then, it verifies that each module is an authorized module and that its integrity is preserved. To check the integrity of the modules,

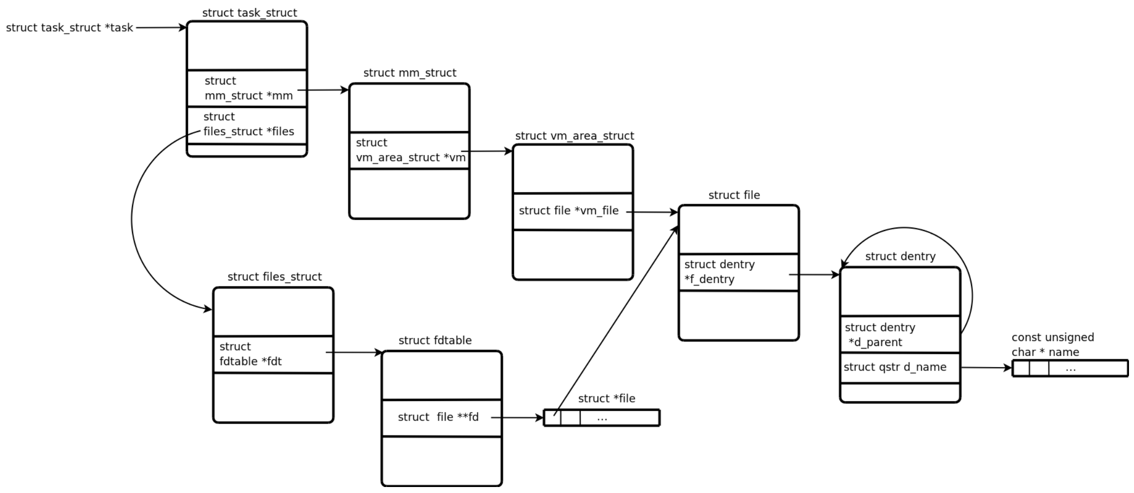


Figure 5.4: Rebuilding File Data-Structures through the Introspection Library

before starting the Mon-VM, each authorized kernel module is loaded and the I-VM computes the hash of the pages storing its instructions and saves these values along with the name of the module. Later, when this Mon-VM is started, this function periodically computes the hash of the pages storing the code of each kernel module and it checks if the hash differs from the previous one. If a hash or the name of a module differs from the stored values, then either a module has been modified or a not authorized one has been loaded.

5.2.2.5 Promiscuous Mode Checker

This function requests the pages starting from the kernel symbol `dev_base`, a pointer to a list of device structures in the Mon-VM. For each of such structures, this function checks the corresponding flags to discover whether the interface is set into promiscuous mode. This approach is similar to the one implemented by Kstat [88], but it differs because of the level where this check is applied. Kstat accesses these structures at the user-level through `/dev/kmem` or, if implemented as a module, at the kernel-level. PsycoTrace promiscuous mode checker function applies the same checks at the VMM level, so it cannot be defeated even by an attacker that gains root privileges.

5.2.2.6 Anti-spoofing

To support the anti-spoofing capabilities, the I-VM kernel is compiled with the following options:

- `CONFIG_NETFILTER_XT_MATCH_PHYSDEV`
- `CONFIG_BRIDGE_NETFILTER`

- `CONFIG_NETFILTER_NETLINK`
- `CONFIG_NETFILTER_XTABLES`
- `CONFIG_BRIDGE`

The I-VM implements the anti-spoofing checks on the Xen virtual bridge using a set of iptables [167] rules. Each rule is defined in terms of the static IP address bound to the virtual interface assigned to the Mon-VM. Every packet with a spoofed source IP address is dropped and logged.

5.2.3 Context-Agent Injection

The I-VM can inject a context-agent inside a Mon-VM at any time and it also implements mechanisms to control its behavior and to communicate with it to exchange commands and results. The I-VM also protects the context-agent from modifications by any software running in the Mon-VM (see Fig. 5.5).

The I-VM has to provide a trusted subset of the Mon-VM kernel through the mechanisms previously described: this is a necessary condition, because the context-agent relies on the integrity of some key-components on the Mon-VM kernel so that it may exploit the kernel interface to properly perform its tasks. The approach that PsycoTrace adopts to deal with this condition is described in the next section.

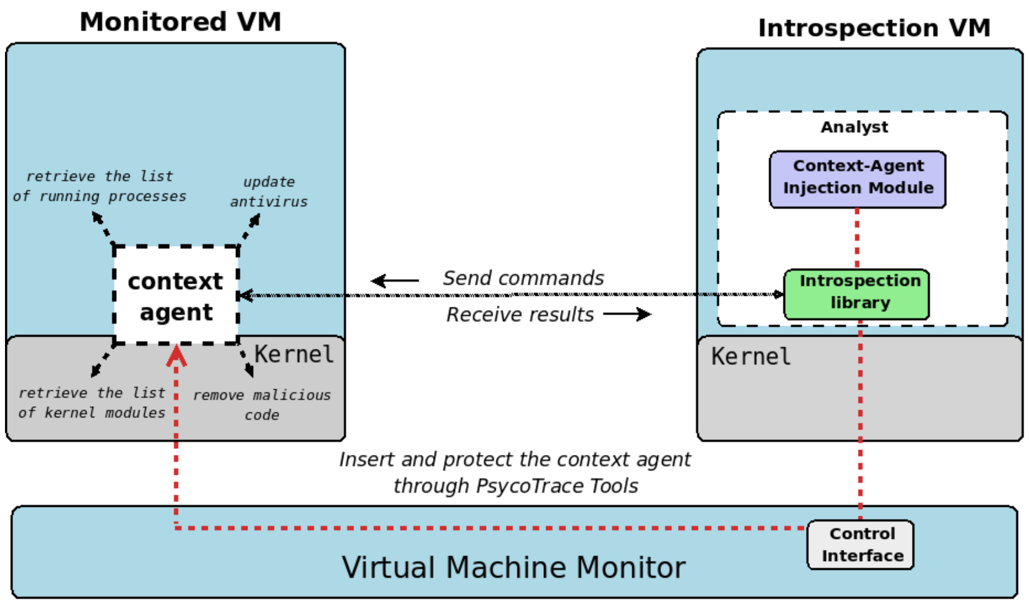


Figure 5.5: Context-Agent

5.2.3.1 Defining the Kernel Trusted Computing Base

The algorithm to inject a context-agent heavily depends on the assumption on the Mon-VM kernel components that can be trusted. There are three cases to consider:

1. the kernel is untrusted;
2. the kernel is fully trusted;
3. the kernel is partially trusted because the I-VM checks the integrity of key kernel components through VMI.

If the kernel is untrusted, then the context-agent should also include some low kernel-level routines, such as to allocate memory, and some kernel data-structures to act independently from the Mon-VM OS. It is worth stressing that there is no difference between this case and a pure VMI approach where the I-VM gets information on the Mon-VM without injecting the context-agent. In fact, in both cases the OS-view has to be rebuilt starting from raw data, because any information that may be supplied by the kernel cannot be trusted. For these reasons, the implementation of this solution is rather complex and its cost effectiveness is rather low.

If the kernel is fully trusted, a possible approach is the one that hijacks the execution of a root process and inject code into its memory to execute a user-level program (such as `insmod`) to load a kernel module that implements the context-agent. Before injecting the context-agent, the I-VM should force the hijacked process to mount the file-system that stores the kernel module. Alternatively, if the context-agent is implemented through a user-level program, the I-VM should inject code into the hijacked process on the Mon-VM to execute a `fork()` and then an `exec()` to load and run the context-agent. Even if the complexity of this solution is rather low, it is based upon a strong assumption on the kernel, i.e. that it is fully trusted.

The approach that we have implemented belongs to the case where only a subset of the kernel can be trusted. In this way, the I-VM can:

- exploit PsycoTrace kernel integrity function to determine a minimal subset of the Mon-VM kernel that can be trusted;
- after providing a trusted subset of the kernel, the I-VM can inject into the Mon-VM a kernel module, which implements the context-agent and may safely invoke the trusted kernel functions.

There are two alternative implementations of this approach:

1. inject the kernel module into *user-space memory*: the I-VM should mimic the behavior of `insmod`, by forcing the Mon-VM to invoke a function or a system call to reclaim some user-space memory and by overwriting the allocated memory with the content of the module. Finally, the I-VM should invoke

the Mon-VM kernel system call to load a kernel module from the allocated memory;

2. inject the kernel module into *kernel-space memory*: the I-VM should force the Mon-VM to invoke a kernel internal function to reclaim some kernel memory and it should replicate the functionalities of the system call to load a kernel module by properly modifying the Mon-VM memory and the CPU registers.

In the next paragraphs, we discuss the pros and cons of both approaches.

User-Space Memory Injection. The following are some considerations about injecting a kernel-module into Mon-VM user-space memory:

- firstly, the I-VM has to choose a suitable user-process to hijack, since it needs to allocate user-level memory that is bound to a specific user-process;
- the I-VM has to overwrite the hardware state to issue a memory allocation system call or to invoke a LIBC function, i.e. a `brk()` system call (the system call used to reclaim more memory by incrementing the size of the data segment) or `malloc()` (a LIBC wrapper for `brk()`). In the latter case, the hijacked process should have the LIBC mapped in its address space, and the I-VM should locate the address of `malloc()`. Instead, to invoke the system call `brk()`, it needs to change the hardware state only, such as the general-use registers `EAX` and `EBX`, which are some of the registers exploited by system calls;
- the I-VM needs to overwrite both the Mon-VM memory with the context-agent and the processor's registers to issue the `sys_init_module()` system call, so that the Mon-VM OS can load the kernel module implementing the context-agent. The first parameter of the system call is the address of the memory previously allocated;
- as soon as the `sys_init_module()` returns, the I-VM should restore the previous state of the hijacked process.

In an alternative solution, the I-VM forces a Mon-VM user-process to issue a `fork()`, so that the I-VM can apply the algorithm with a new process that it can tamper with. The problem posed by this approach is that Linux adopts the copy-on-write strategy, so the father and the children may share some memory, and the I-VM may write also into the memory of the father. Another problem is related to the paging mechanism, i.e. if the I-VM has to overwrite some user-space pages that are currently not in memory.

Kernel-Space Memory Injection. The following considerations about overwriting kernel-space memory hold:

- the I-VM only needs to invoke `kmalloc()` to reclaim some Mon-VM kernel memory or simply overwrite an unused kernel memory with the context-agent kernel module;
- the I-VM needs to mimic the `sys_init_module()` system call inside the Mon-VM, which could be can be very hard to emulate.

Since kernel-space memory injection poses too many problem, the first prototype of context-agent injection applies user-space memory injection, by hijacking a Mon-VM user-space process. Basically, the I-VM executes the following steps:

1. freezes the execution of the Mon-VM OS;
2. hijacks a root-owned user-level process;
3. modifies the Mon-VM virtual hardware to invoke a memory allocation function and to transmit to the function the arguments to allocate a buffer that can store the kernel module;
4. copies the kernel module into the allocated memory;
5. invokes the Mon-VM kernel system call to load the module.
6. restores the Mon-VM CPU and stack states to those at step 1.

In the next section, we describe this solution in detail.

5.2.3.2 Algorithm to Inject the Context-Agent

As previously said, in this approach the I-VM mimics the behavior of the `insmod` inside the Mon-VM. The following are the implementation choices that we have adopted:

- the I-VM hijacks the Mon-VM's `init` process because:
  - it is always present on any Linux system;
  - it is a root-owned process that can invoke any privileged system call, such as `sys_init_module()`, and lock any region of memory;
  - its PID is fixed to 1, so it is easy to find the corresponding PCB.

- to store the kernel module, the I-VM allocates user-memory into the Mon-VM through `brk()` rather than through `malloc()`. The reason is that `brk()` is a system call and, hence, the I-VM needs not to locate the address of the `malloc()` function inside LIBC[2]. The I-VM invokes `brk()` twice: firstly to retrieve the current pointer of the data-segment of `init`, which is the memory region that will hold the context-agent, and secondly to move the pointer to reclaim more memory;

- the I-VM modifies both the Mon-VM status and `init` to invoke `sys_init_module()` to load a kernel module: the parameter for this call is the pointer to the `init`'s user-memory previously allocated with `brk()` and overwritten with the context-agent;

- to solve the paging problem, the I-VM locks in Mon-VM memory the pages that will store the context-agent through the `lock()` system call;

- the I-VM exploits `brk()` also to free the reclaimed memory by setting the data-segment pointer to the original value: for the sake of simplicity, we refer to this operation as `free()`.

To implement these operations, the injection of the context-agent requires that a "prequel code", hereafter *loading-code*, is firstly injected into the Mon-VM's `init` process memory to invoke, in order:

```
brk();              //to reclaim some user-space memory
mlock();            //to lock the reclaimed memory
sys_init_module();  //to load the context-agent (a kernel-module)
munlock();          //to unlock the reclaimed memory
free();             //to release the memory
```

Moreover, the I-VM sets triggers on the *loading-code* so that in-between the execution of steps 2 and 3 the I-VM overwrites the allocated memory with the kernel module that implements the agent, hereafter called the *context-agent*[3].

In more detail, the I-VM implements the following steps to inject the *context-agent* (see Fig. 5.6):

1. wait that the Mon-VM OS is fully loaded; then, the I-VM invokes an introspection function to:

    - retrieve the address of Page Global Directory (PGD), i.e. the pointer to the page tables, of the `init` process in the Mon-VM[4];

[2]A further problem is that LIBC can be either statically compiled or dynamically loaded.

[3]In the current implementation, the kernel module is compiled into the Mon-VM and its object `.ko` file is then copied into the I-VM: in a real environment, the kernel module should be compiled in a testbed Mon-VM with the same version of kernel of the hijacked Mon-VM.

[4]The I-VM fetches the PGD address of the Mon-VM's `init` process because its has to translate

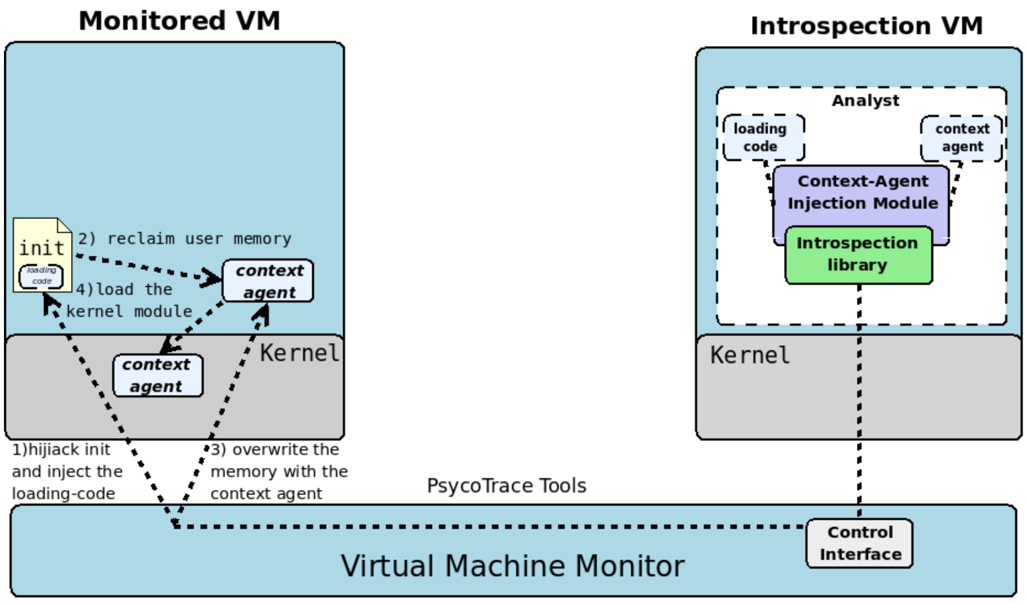


Figure 5.6: Injecting a Context-Agent

- set triggers with each page of `init`'s code address space.

From this moment on, the I-VM waits for `init`'s triggers to be invoked so that it can overwrite its memory with the *loading-code*;

2. the first time `init` generates a trigger, the I-VM saves the content of the `init`'s memory pointed by the current PC and the current Mon-VM CPU registers to be able to restore them. Then, it overwrites the `init`'s memory with the *loading-code*. From this moment on, the I-VM checks whether the trigger belongs to a known offset inside the *loading-code*;

3. it checks if the trigger address is equal that of the `sys_init_module()` instruction inside the *loading-code*: in this case, the I-VM overwrites the `init`'s reclaimed memory with the *context-agent*[5];

4. if the *context-agent* shares a page to store results and/or receive commands, firstly it communicates the address of the shared page through a predefined kernel variable[6];

the `init`'s code page addresses into physical ones to associate them with triggers. The I-VM retrieves the starting address and the size of the `init`'s code segment from the Mon-VM `proc` file systems through introspection.

[5]Since the I-VM does not know the address of the `init`'s user buffer that stores the *context-agent*, it retrieves this value from the Mon-VM `EBX` register: in fact, `EAX` stores the system call number for `sys_init_module()`, while `EBX`, `ECX`, `EDX` store the values of the three parameters for this call.

[6]The I-VM checks if the trigger address is that of the predefined kernel variable. If so, it

5. it checks the trigger address is that of the synchronization variable; as soon as the *context-agent* has updated the shared page, it changes the value at this location to signal the I-VM that it may access the shared page to retrieve the results of the checks: in the testbed examples, the shared page stores the list of running processes, or the list of open files, or the list of loaded modules;

6. it checks if the trigger address is that of the last instruction of the *loading-code*: in this case, the I-VM restores the state of memory of `init` and Mon-VM CPU registers to the values they hold before the hijacking.

The implementation of this algorithm results in an important contribution of this framework: the injection of the context-agent into the Mon-VM does not require the cooperation of the Mon-VM and *everything is transparently applied from the I-VM.*

Appendix B contains a technical description of the algorithm implemented by the I-VM to retrieve from the Mon-VM memory (i) the PGD address of the `init` process; (ii) the code of the context-agent to check that it has not been attacked.

5.2.3.3 Algorithm to Remove the Context-Agent

Basically, the code to remove the *context-agent*, called the *delete-code*, implements the same strategy of the *loading-code*. The only difference is that instead of invoking the system call to load the kernel module, the *delete-code* invokes the system call to remove the kernel module. Hence, the I-VM injects the *delete-code* into the `init` process memory to invoke, in this order, the system calls:

```
brk();                //to reclaim some user-space memory
mlock();              //to lock the reclaimed memory
sys_delete_module();  //to unload the context-agent (a kernel-module)
munlock();            //to unlock the reclaimed memory
free();               //to release the reclaimed memory
```

To this end, the I-VM sets triggers on the *delete-code* so that in-between the execution of steps 2 and 3 the I-VM overwrites the allocated memory with the name of the *context-agent*: in fact, this memory is exploited to store the parameter used by the system call `sys_delete_module()` to remove a kernel module.

In more detail, the I-VM implements the following steps to remove the *context-agent* from the Mon-VM's memory:

1. as soon as `init` generates a trigger, the I-VM saves the content of the `init`'s memory pointed by the current PC and the current Mon-VM CPU registers to be able to restore them. Then, it overwrites the `init`'s memory with the *delete-code*. From this moment on, the I-VM checks whether the trigger belongs to a known offset inside the *delete-code*;

retrieves the address of the shared page and associates it with a trigger.

2. if the trigger address is that of the `sys_delete_module()` instruction inside the *delete-code*, the I-VM overwrites the `init`'s reclaimed memory with the name of the *context-agent*[7];

3. if the trigger address is that of the last address of the *delete-code*, the I-VM restores the status of the memory of the hijacked `init` process and of the Mon-VM CPU registers to their original status.

5.3 Checking the Process Self

Once the integrity of the Mon-VM kernel is assured through a cooperation between PsycoTrace's integrity functions and the context-agent, the run-time support has to guarantee that the self of P is not altered. To do this, PsycoTrace's run-time tools trace and check system calls against $CFG(P)$ and evaluate invariants in $IT(P)$.

5.3.1 System Call Tracing

At run-time, system calls are traced by two tools: the *HiMod* and the *Analyst*, which run, respectively, in the kernel of the Mon-VM and as a user-process in the I-VM (see Fig. 5.1). The HiMod is a kernel module that hijacks the system calls that P invokes. Every time P invokes a system call, HiMod notifies the Analyst. The Analyst includes a Bison-generated parser for $L(P)$, which checks the trace of P, and an Assertion Checker, which evaluates invariants by exploiting the Introspection Library to access the memory of P and the Mon-VM VCPU registers. The interactions between the Analyst and the HiMod are synchronous. The HiMod traps the system calls of P and, before servicing the trapped call, it informs the Analyst that is waiting for communications from the HiMod. Then, the Analyst freezes the execution of the Mon-VM and resumes this execution only after successfully terminating the security checks.

To notify the Analyst that P wants to issue a system call, the HiMod allocates and shares with the Analyst an *event channel*, which is a data structure that Xen introduces to emulate the interrupt mechanism. When the Analyst allocates a new event channel, it receives an integer value that represents the port number used to capture notifications from the HiMod. At this point, the Analyst binds itself to the specified port and waits for a notification from the HiMod. Each notification corresponds to a system call invocation that P wants to issue. Because of the large number of system calls in the Linux kernel, and since most of them cannot be

[7]The I-VM exploits VCPU-Introspection to retrieve this memory address from the `EBX` register, which stores the first parameter of the `sys_delete_module()` system call, i.e. the name of the module to be removed.

exploited to attack a process, PsycoTrace monitors only the system calls listed in Tab. 5.1, which are critical from the security point of view [30].

sys_exit	sys_mknod	sys_setfsgid	sys_setfsuid	sys_read
sys_chmod	sys_lchown	sys_setresgid	sys_write	sys_vhangup
sys_symlink	sys_mkdir	sys_open	sys_stat	sys_chown
sys_ioctl	sys_close	sys_lseek	sys_setgid	sys_ftruncate
sys_waitpid	sys_getpid	sys_setgroups	sys_flock	sys_creat
sys_mount	sys_setresuid	sys_brk	sys_link	sys_fchown
sys_rename	sys_reboot	sys_unlink	sys_setuid	sys_fchmod
sys_swapoff	sys_chdir	sys_setregid	sys_setreuid	sys_stime
sys_delete_module	sys_mlock	sys_settimeofday	sys_setdomainname	sys_truncate
sys_setrlimit	sys_ioperm	sys_sched_setparam	sys_swapon	sys_mlockall
sys_nice	sys_sethostname	sys_socketcall	sys_syslog	sys_rmdir
sys_dup2	sys_nfsservctl	sys_kill	sys_setpriority	sys_adjtimex
sys_umount	sys_sysctl	sys_sched_setscheduler	sys_quotactl	sys_exec
sys_time				

Table 5.1: Traced System Calls

As soon as the Analyst is notified that P has invoked a system call, it suspends the Mon-VM and retrieves the values of the processor's registers in the Mon-VM through VCPU-Introspection. In this way, the Analyst knows the system call number and, if needed, its parameters and the values of any program variable. After retrieving the system call number from the `EAX` register, the Analyst passes it to the lexical analyzer that, in turn, transmits the proper system call token to the parser. If the current call does not belong to the terminal alphabet symbols, i.e. it is a system call that P should not invoke, the parser returns an error. Otherwise, it checks the current system call by resuming the parsing from the point reached when analyzing the previous call. No error is signaled if, after receiving the current token, the parsing continues till it requires a further token that corresponds to the next system call of the process. As a matter of fact, this implies that the current trace of calls may belong to at least one legal trace in $CFG(P)$. In this way, PsycoTrace implements a *stream-oriented parsing* as opposed to the usual application of a parser to a whole sequence of tokens in a single step. This strongly simplifies the parsing with respect to the case where a new derivation from the starting symbol of the grammar is started from each invocation. The corresponding performance improvement favors the adoption of a context-free grammar and of a Generalized Left-to-right Rightmost derivation parser (GLR) parser.

5.3.2 The Analyst

The *Analyst* implements the run-time checks to verify the integrity of the self of P and is composed of:

- *Lexical Analyzer*: it verifies that the system call that P wants to issue belongs to the set of system calls returned by the static analysis of $SourceCode(P)$;

- *Parser*: it checks that the current trace of system calls issued by P is coherent with $CFG(P)$, i.e. it is a prefix of a word allowed by $CFG(P)$;
- *Assertion Checker*: it checks whether the invariant coupled with the current system-call holds.

5.3.2.1 Lexical Analyzer

The lexical analysis of the tokens of $L(P)$ only verifies that every system call that P attempts to invoke belongs to the set of system calls it may legally invoke, which represent the valid tokens of $L(P)$. In this scenario, we can apply a tool to automatically generate a scanner, such as Flex or YooLex, that accepts a description of the language tokens through regular expressions. In this way, the automatic generation of the lexical analyzer for $L(P)$ is parametric to the system calls that P may invoke. An example of an input for Flex is the following one:

```
%%
open { return(OPEN); }
getuid { return(GETUID); }
exit { return(EXIT); }
[ \t\n]+ [a-z]* { return("ERROR"); }
%%
```

In this example, Flex generates a scanner that only accepts the tokens in {`open, getuid, exit`}, and it returns a lexical error otherwise. The lexical analysis function `yylex()` is automatically generated by the tool that builds $CFG(P)$ after a visit of $AST(P)$ that returns the system calls that P may invoke. The lexical analyzer is then included in the prologue of Bison, so that the `yylex()` function can be invoked before performing any syntactic analysis. Also this procedure is parametric with respect to the system calls and it can be automatically generated during the static analysis of *SourceCode*(P). Analogously, by adopting a scanner generator, it is possible to automatically generate the set of previous regular expression. An example of a scanner automatically-generated is shown in Tab. 5.2, where P can only execute three system calls: `open()`, `write()` and `execl()`.

5.3.2.2 Parser

A system call invocation does not violate the self of P if the trace of system calls generated up to a given call is a prefix of at least one string generated by $CFG(P)$. A Bison-generated parser implements the syntactic analysis to verify that the execution flow of the process expressed in terms of system calls, is coherent with the self defined by $CFG(P)$. The Analyst and the parser are executed as two separated processes on the I-VM. The Analyst reads the system call number from the processor's registers and passes the token to the parser that invokes the function

```c
. . .
//LEXICAL ANALYZER FUNCTION CODE AUTO-GENERATED!
int yylex(void) {
   char *sys_tokens[NSYSCALL] = { "open", "write", "execl" };
   char *sys_addr_sep[2];
   char sys_addr_received[35];
   int c, i;
   //Skip white space.
   while ((c = getchar ()) == ' ' || c == '\t');
      if(setup_token) {
         //Process setup message.
         scanf("%s", sys_addr_received);
         setup_token = 0;
      }

   //Process syscall.
   scanf("%s", sys_addr_received);
   split(sys_addr_received, sys_addr_sep, 2, "-");
   for(i = 0; i < NSYSCALL; i++) {
      if(strcmp(sys_addr_sep[0], sys_tokens[i]) == 0) {
         switch(i) {
            case(0): return OPEN; break;
            case(1): return WRITE; break;
            case(2): return EXECL; break;
         }
      }
   }
   //Return end-of-input.
   if (c == EOF)
   return 0;
}
```

Table 5.2: PsycoTrace Scanner for P

`yylex()` before performing syntactic analysis. The Analyst kills P if the system call is not coherent with $CFG(P)$. Grammars corresponding to the simple constructs shown in Tab. 4.1 generate a language that is accepted by a deterministic top-down lexical analyzer. In other words, these are LL(1) grammars. This simplification is possible because the code fragments shown in Tab. 4.1 are outside the real context of the program. In fact, if we apply GGA to more complex program, it may generate productions that are quite similar to the following ones (shown using the Bison syntax):

```
S0: TIME CLOSE DUP2 DUP2 DUP2 F0 | TIME CLOSE DUP2 DUP2 DUP2 S0
;
F0: READ F1 CLOSE | READ F1 WRITE F0
;
F1: F2 | /* empty */
;
F2: OPEN WRITE F2 CLOSE | OPEN WRITE CLOSE
;
```

Since a static analysis cannot predict which branches in the code will be executed, to correctly identify the process self, $CFG(P)$ includes all the possible productions. In this scenario, non-determinism is resolved by adopting the Bison-generated GLR Parser. The high complexity of parsing, due to ambiguous context-free grammars, is justified by a better accuracy of checks also for programs with a highly non-deterministic behavior.

5.3.2.3 Assertion Checker

The proposed semantics-driven integrity measurements also include the evaluation of invariants. System call sites are one of the most appropriate choices for invariant evaluation, because at these points the monitored system switches from user-level to kernel-level. To be fully integrated with the run-time tool, PsycoTrace static tool is focused on, but not restricted to, the generation of invariants that relates values of programs variables and of system call parameters, where each invariant is coupled with the virtual address of the corresponding system call. By coupling an invariant with each call, PsycoTrace can detect non-control-data attacks [48, 45].

The I-VM runs an *Assertion Checker* to evaluates invariants and, even if it can monitor several processes concurrently, for the sake of simplicity, we assume that P is the only process that is being monitored.

In the current prototype, the input of the Assertion Checker is a set of invariants of the form:

```
(PC, var name: addr: type, expr on vars)
```

where:

- `PC` is the program counter, i.e. the virtual address, of a system call;
- `var name: addr: type` is a set of variable names, their virtual address and their type;
- `expr on vars` is a set of relations among variables with the following structure:
  - `<var (OP var)* REL value >`, where `OP` is an arithmetic/logic operator and `REL` is a relational operator, such as: `a > 10;`, `a + b >= 0;`, `i == 5;`
  - `<var (OP var)* REL var >`, such as: `a + b > c;`, `c == d`.

As an example, let us consider the invariant:

`( i, {a:0xB7EC00DA:int}, {a == 5} )`

The following semantic action is coupled with the invariant of the Assertion Checker at the execution point `i` equal to `0xB7EC00DA` of P:

```
if(!((int)(*map(0xB7EC00DA)) == 5)) {
//signal that an invariant does not hold here
//kill P
...
}
```

where `map` is the function that maps the virtual addresses into the address space of P. Suppose that, in this example, the point of execution `i` (i.e., `0xB7EC00DA`) corresponds to the invocation of the first terminal `DUP2` in production `S0` of the previous grammar. In this scenario, the following grammar is generated with the annotated semantic actions:

```
S0: DUP2
{
   if(!((int)(*map(0xB7EC00DA)) == 5)) {
   //signal that an invariant does not hold here
   //kill P
   ...
   }
} DUP2 { ... } DUP2 { ... } F0 { ... }
;
...
```

Any non-empty assertion is the conjunction of assertions in the following classes:

1. *Parameters assertions.* They express data-flow relations among parameters of distinct calls, e.g. the file descriptor in a read call is the result of a previous open call.

2. *File Assertions.* To prevent symlink and race condition attacks, they check, as an example, that the real file-name corresponding to the file descriptor belongs to a known directory.

3. *Buffer length assertions.* They check that the length of the string passed to a vulnerable function is not larger than the local buffer to hold it.

4. *Conditional statements assertions.* They prevent problems due to impossible paths [243] by relating a system call and the expression in the guard of a conditional statement. As an example, in `if(uid == 0) then syscall`$_1$ `else syscall`$_2$, we couple the assertion `uid == 0` (usually, on many OSes, this is the user-id of root) with `syscall`$_1$, to check at run-time the real value of `uid` to prevent a normal user from executing the same call with the privileges of the root user. They may also check that the current return address matches the call issued by P.

Currently, PsycoTrace implements these classes for the purpose of prototyping: obviously, they are not comprehensive of all the critical problems for an application. According to the default-deny approach, the handling of a false invariant, is implemented through the invocation of the error recovery function `yyerror` that implements the error reporting and kills P.

Invariant Evaluation. To evaluate the invariants, the Assertion Checker exploits the VCPU-Context introspection capability of the library to retrieve the current PC of P and to map the pages storing the variables of P into its address space to fetch their values.

Whenever a system call is issued, the Assertion Checkers needs to retrieve the address of the instruction currently executed by P to locate the invariant coupled with the system call in the Invariant Table. Instead of fetching the value of PC, the current implementation retrieves the system call's return address that is, more precisely, the system call handler's return address. This solution has been adopted because this return address points to the instruction following the system call site (i.e., the address following the PC coupled with the system call) and, therefore, can be easily related to the system call address returned by the static analysis and coupled with the invariant. This address is located in the user-stack but, since after the invocation of the current system call the Mon-VM is in kernel space, the `ESP` register points to the kernel stack not to the user stack. Thus, the Assertion Checker needs to retrieve the value of the saved `ESP` register in the kernel stack to retrieve the return address. Then, from the user-stack it locates the return address.

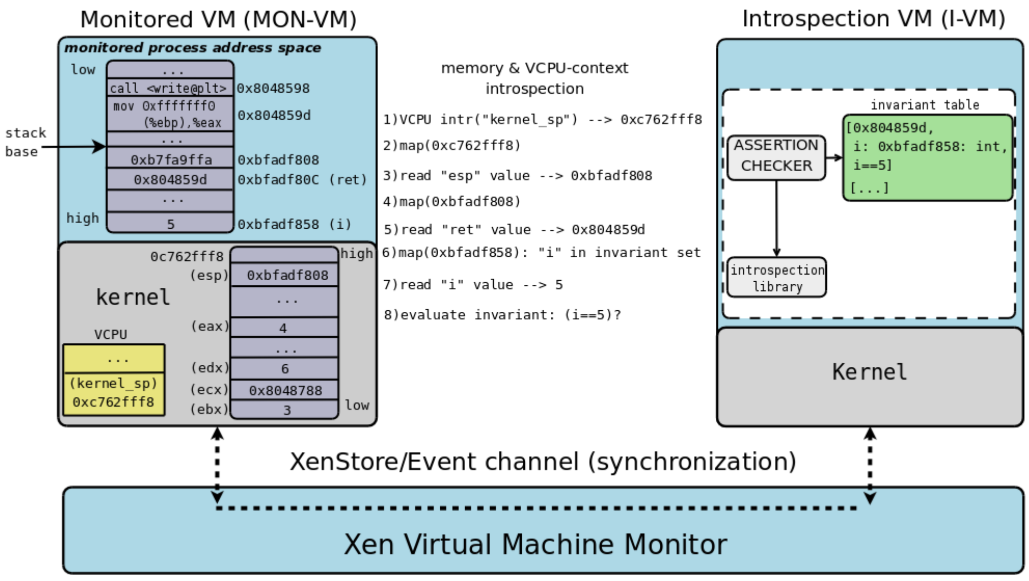


Figure 5.7: Run-Time Invariant Evaluation

In more detail, to evaluate an invariant the Assertion Checker implements the following steps (see Fig. 5.7):

1. accesses the VCPU context to read the `kernel_sp register`, which points to the top of the kernel stack;
2. maps in its memory the kernel stack;
3. reads the value of the `ESP` register, which points to the base of the user stack;

4. maps in its memory the user stack;
5. locates the return address of the system call in the user stack of P. Since the offset of the return address from the stack pointer depends upon the system call type, the Assertion Checker reads the `EAX` register to identify the system call;
6. after reading the return address, the Assertion Checker in its memory maps the pages storing the variables coupled with this return address;
7. reads the value of the variables;
8. evaluates the invariant.

If the invariant is satisfied, the Assertion Checker resumes the execution of the Mon-VM, otherwise it kills P.

5.3.3 Sliced Execution

By exploiting VMI, PsycoTrace can compute the values of the assignment values that do not not satisfy the conditions of Sect. 4.3.2. This can be done by shifting the execution of some parts of the program of P from the Mon-VM to the I-VM to compute, at run-time, the values of the assignment values that cannot be determined inside a code fragment. At the entry point of a code fragment, we can compute a slice that corresponds to the assignment variable that cannot be determined and move this section of code inside the I-VM. The I-VM will execute this code, to compute the values of the variables involved in the invariant. As an example, in the following code:

```
...
char *filename;
L0: uid_t uid = getuid();
L1: filename = function(argv[1]);
L2: Log(...);
L3: execl(filename, NULL);
...
```

`filename` is an assignment variable (i.e., $k =$ `filename`) that cannot be determined when `getuid()` is executed at `L0`, because the three conditions of Sect. 4.3.2 are not satisfied. In fact, in the system block that begins at `L0` (i.e., $i =$ `L0`) and ends at `L3` (i.e., $o =$ `L3`), the third condition is not satisfied since at `L0` the function `function()` updates the assignment variable k in-between i and o. But, by exploiting the slicing technique, we can compute the slice corresponding to `filename` at the execution of `getuid()`. The slice contains line `L1` and all the other lines to compute the final value of `filename`. Therefore, by moving the slice in the I-VM, the value of `filename` can be computed so that it is available during the evaluation of the invariant coupled with `execl()` (at `L3`) as previously described (see Fig. 5.8).

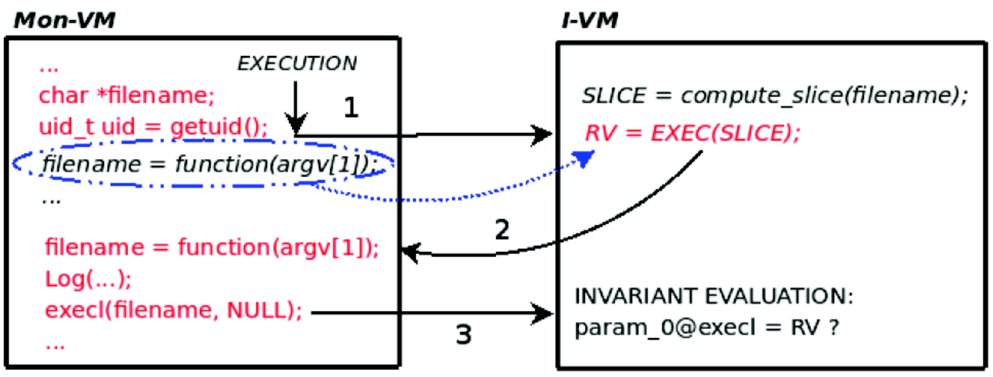


Figure 5.8: Sliced Execution

5.4 Results

This section presents a first evaluation of both PsycoTrace kernel integrity functions and run-time tools to check the process self, from the security and the performance points of view.

5.4.1 Protecting the Kernel Integrity

To evaluate the kernel integrity functions, we configured the I-VM to compute the hashes of the text area of the kernel and of the kernel modules with a predefined frequency. In turn, a context-agent into the Mon-VM kernel returns the list of running processes, which the I-VM compares against the equivalent list returned by the Running Processes Checker function using introspection. The I-VM also verifies that only authorized kernel modules have been loaded and also checks that the Mon-VM is not sniffing traffic. Lastly, it applies anti-spoofing techniques on the virtual bridge. The Introspection Library and the PsycoTrace's kernel integrity functions are composed of about 1.5K lines of C code, whereas the context-agent framework consists of about 2.5K lines of C/ASM code.

5.4.1.1 Effectiveness

To evaluate the effectiveness of the kernel integrity functions, we have modified some known rootkits to update the kernel text and an entry in the `idt_table` pointing to a modified interrupt handler [130]. We also inserted a malicious module into the kernel of the Mon-VM to modify the `sys_call_table` and an existing system call. Besides, we replaced system binaries, such as `ps`, to hide specific processes. The I-VM correctly detects the modifications to:

- the system call handler;
- any system call;

- pointers in the `sys_call_table`;
- an entry in the `idt_table`.

Lastly, each time a module is loaded into the kernel, the I-VM detects if the module is not an authorized one, because it does not belong to the list of know modules, or if an authorized module has been updated.

5.4.1.2 Performance Evaluation

To evaluate the overhead due to kernel integrity checks, we computed the average time of 100 consecutive executions of the command `tar -xjf linux-2.6.20.tar.bz2` on a Mon-VM, while the I-VM applies the whole set of consistency checks previously discussed, with a period of 60 seconds between each invocation. This is a worst case since the `tar` command is computing intensive and it is not possible to overlap the checks with some I/O activity. The overhead is less than 10% with respect to the execution of the same command when the integrity checks are not applied.

5.4.2 Checking the Process Self

This section analyzes the attacks against P that PsycoTrace can detect and it shows a first evaluation of the run-time overhead of the current implementation. PsycoTrace run-time tools are implemented through 450 lines of Perl code, which generate the HiMod by parsing the definition of the Linux system calls. The HiMod consists of 2.5K lines of C code, while the Analyst is composed of about 3K lines of C code. Finally, the generated Bison parser for $CFG(P)$ is about 2.5K lines of C code.

```
<MAIN>: "dup2" "dup2" "dup2" ("read" <PARSE_STR>
        ("write")?)* "close".

<PARSE_STR>: (<LOGFILE>)?.

<LOGFILE>: "open" "write" (<LOGFILE>)? "close".
```

```
S0: DUP2 DUP2 DUP2 F0;
F0: READ F1 CLOSE |
    READ F1 WRITE F0 CLOSE;
F1: /* empty */ | F2;
F2: OPEN WRITE F2 CLOSE |
    OPEN WRITE CLOSE;
```

Table 5.3: Context-Free Grammar for P and its Bison Representation

5.4.2.1 Effectiveness

To test the effectiveness of PsycoTrace, we considered a case where P implements a single server application that opens a socket and reads from its stream a sequence of characters. Appendix C contains $SourceCode(P)$ used for the tests. Table 5.3

shows the $CFG(P)$ generated by the GGA and the corresponding grammar in Bison syntax. Semantic actions are not shown. For the sake of conciseness, $CFG(P)$ only describes the behavior of P after the `accept()` system call. Notice that the `LOGFILE` non-terminal generates a recursive production which defines the language $(\texttt{open write})^n(\texttt{close})^n$, which cannot be handled by a regular grammar. The `parse_str()` function parses the received string. If the string begins with "copy", P invokes `strcpy()` to copy the receiving string into a local small buffer. `strcpy()` is an insecure function that could be exploited to compromise the security of the application. If, instead, the received string begins with "file", P invokes the `logfile()` function. Lastly, if the string begins with "exit" the server closes the connection with the client. The following is one of the traces generated by the execution of P:

$\texttt{(DUP2)}^3\texttt{; READ; (WRITE; READ; (OPEN; WRITE)}^{10}\texttt{; (CLOSE)}^{10}\texttt{)}^2\texttt{; (WRITE; READ)}^2\texttt{; CLOSE.}$

We have implemented an attack that exploits the vulnerable `strcpy()` function on the server-side, to manipulate a parameter from the client-side. The exploit overflows the server buffer by transmitting a string that contains a shellcode and that is built by appending to the string "copy" a sequence of `NOP` instructions, the shellcode itself and finally a repetition of the jump address of the shellcode to overwrite the `parse_str()` return address in the server stack. The execution of this exploit results in a remote shell with the privileges of the remote server process. The trace of P after a successful attack is:

$\texttt{(DUP2)}^3\texttt{; READ; SETUID; BRK; OPEN; CLOSE; (OPEN; READ; CLOSE)}^3\texttt{; OPEN; CLOSE; (BRK)}^3\texttt{; TIME; BRK; IOCTL; BRK; (OPEN; READ; CLOSE)}^2\texttt{; BRK.}$

In this case, PsycoTrace parser signals an inconsistency with respect to the expected behavior after the fourth system call, P is stopped and no shell is spawned. The corresponding string is:

```
DUP2; DUP2; DUP2; READ; syntax error [SETUID] → process killed (pid=1054).
```

5.4.2.2 Performance Evaluation

The system to run the prototype tools included a Pentium Centrino Duo T2250 1.7GHz. In all the tests, 128MB of physical RAM were allocated to the Mon-VM, running a Linux Debian distribution, and 874 MB RAM to the I-VM. The Xen version was 3.1.0, while the Mon-VM Linux kernel version was 2.6.18-xen.

We evaluated the average time to execute the `bunzip2` tool to uncompress the Linux kernel on the Mon-VM in two cases, during a normal execution and when we only traced the `bunzip2` process. The execution time increased from 19.896 sec to 24.268 sec. The overhead, 21.97%, is rather high in this case because `bunzip` is a tool that invokes system calls at high rate. To optimize introspection of variables mapped into the same page (see Fig. 5.9), PsycoTrace exploits a software TLB that records the pairing among virtual and machine addresses. Before translating a virtual address *va*, the Introspection Library searches the TLB for the virtual address of the page including *va*. If the address is found, then the page is already

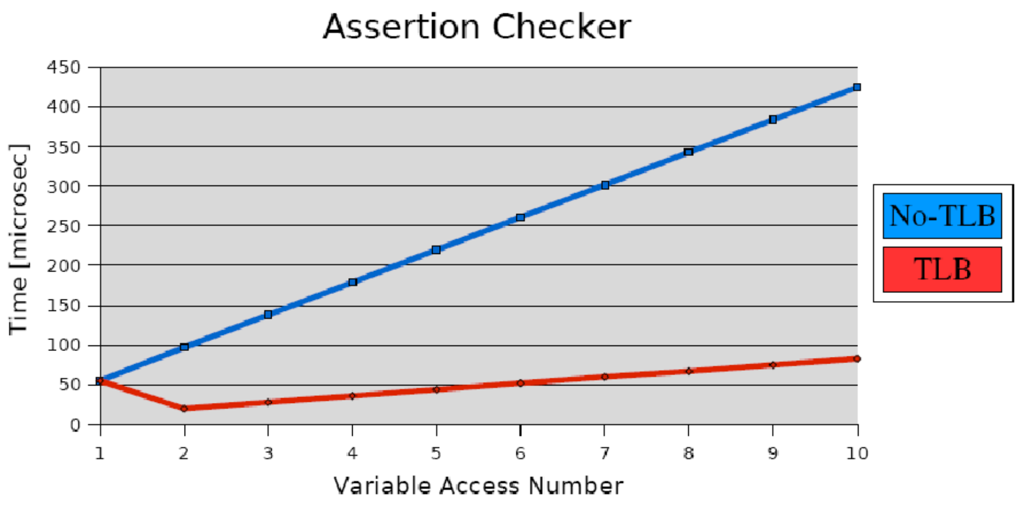


Figure 5.9: Time to Retrieve a Variable Value

mapped in the Assertion Checker memory. The average time to map a page of P into the Assertion Checker address space is about 50μsec. The software TLB reduces the access time to 20μsec anytime the same page stores several variables. In this way, the Assertion Checker can access the variables without mapping further pages of the virtual address space of P.

system call	normal	traced	traced + introspection
time	2 μsec	55 μsec	141 μsec
open	3 μsec	58 μsec	116 μsec
write (1k buffer)	8 μsec	67 μsec	177 μsec

Table 5.4: Overhead of System Calls

Table 5.4 shows the average execution time of three system calls executed on the Mon-VM in three cases: during the normal execution, while tracing the system calls and when the Analyst checks the trace and evaluates the assertions by accessing one page of P. In these tests, to compute the average execution time, the traced program loops several times on each system call.

Fig. 5.10 displays the average execution time of the `time()` system call in a loop when the Analyst evaluates the assertions, as the number of mapped pages varies from 1 to 10. Complex assertions refer to a larger number of pages because they access several variables. The overhead is linear in the number of mapped pages.

Finally, we considered the execution time of P, where $SouceCode(P)$ is described in Appendix C, when a client generates and sends to P a continuous stream of requests. Three cases were analyzed:

1. a normal execution of P;

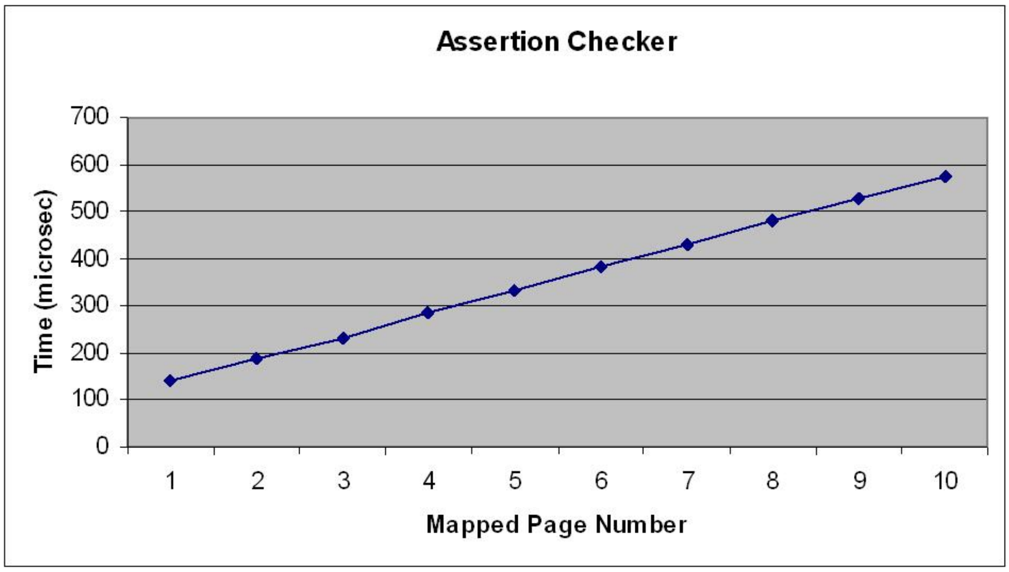


Figure 5.10: Assertion Checker Overhead

2. when system calls generated by P are traced and notified to the Analyst but no check is applied;
3. when tracing the system calls, checking the grammar and evaluating the assertions, by accessing one page of P.

The total number of traced system calls generated by P was 63234. In the worst case, a 48% overhead arises.

Part III

Applications of the Proposed Approach

Chapter 6

Remote Attestation of Semantic Integrity

An *overlay network*, or simply overlay, is a logical and dynamic connection among logical or physical nodes that belong to a predefined pool. In some cases the pool is well known and strictly ruled, e.g. nodes of a corporate network, in other cases it is fully unconstrained, e.g. peer-to-peer (P2P) networks. Overlays are becoming more and more popular because they can offer highly robust services to the nodes they interconnect. As an example, virtual private networks offer confidential communications while P2P networks implement a highly available and distributed data repository. On the other hand, sometimes overlays offer a very low security because almost all these properties are at risk even if a few nodes of the overlay run some malware because of an erroneous, or malicious, configuration or as a result of external attacks against the node. Hence, *node integrity* is a precondition of any overlay security policy and it should be attested not only when a node joins an overlay but also as long as the node belongs to the overlay.

These considerations have led us to define Virtual machine Integrity Measurement System (VIMS), an architecture based upon PsycoTrace to continuously attest the integrity of overlay nodes by applying alternative integrity measurements according to the overlay security requirements. VIMS protects the integrity of a node by defining both a start-up attestation and a continuous monitoring that are applied, respectively, when the node joins the overlay and as long as the node belongs to the overlay. To implement the corresponding measurements, VIMS applies PsycoTrace static tools and extends the run-time ones. In this way, VIMS supports alternative strategies to describe the expected behavior with distinct complexities and attack detection capabilities and it can implement integrity measurements and monitoring strategies that consider not only the correct configuration of an OS and of the applications, but also attacks to install malware.

To strongly separate the attestation subsystem from the one to be attested, VIMS fully exploits the two VMs introduced by PsycoTrace run-time architecture, i.e. the Mon-VM and the I-VM. Henceforth, to stress the role played by the I-VM when supporting the integrity measurement process and to take into account the new components that it runs, we will refer to the I-VM as the *Assurance VM* (A-VM). The only difference between the A-VM and the I-VM is that the A-VM also runs some modules to remotely attest the integrity of the Mon-VM. The Mon-VM is the target of the attestation and of the monitoring and it runs an overlay application that plays the role of the process P protected by PsycoTrace. When VIMS is adopted, each overlay node runs two VMs, the A-VM and the Mon-VM. The A-VM cooperates with the other A-VMs to apply measurements on behalf of the overlay by accessing the live state of the Mon-VM in its node through the PsycoTrace run-time tools. Anytime an A-VM detects a loss of integrity, it kills the Mon-VM on its node and informs the other A-VMs. Trust in VIMS measurements requires the correct configuration of both the A-VM and the underlying VMM and it is guaranteed by measurements and controls of a Trusted Platform Module (TPM) [22, 180] subsystem that acts as the root-of-trust for the chain of measurements.

6.1 Virtual Machine Integrity Measurement System Architecture

VIMS is aimed at implementing a fairly general and reliable system to measure the integrity of an overlay node, so that other nodes can be assured of the integrity of the node. The main goals of VIMS are:

1. granular checks on the integrity of a node willing to join the overlay: with respect to solutions that only exploit TPM-based functions, the integration of static and dynamic tools results in more granular checks;

2. support for dynamic security policies: as long as a node belongs to an overlay, its run-time state should be continuously monitored (either interval-based or on request), to detect whether it has been infected by a malware. Moreover, the security policy that is applied, i.e the PsycoTrace strategy to describe the process self of the overlay application, can be updated as a result of changes in the configuration of a node;

3. mutual attestation: if required, all the parties should be mutually assured of the integrity of any other peer;

4. scalability: the overhead of an attestation should be negligible given the security policy of interest.

As previously said, VIMS extends the set of PsycoTrace run-time components and also defines a protocol that rules both the information exchanged among these components and the format of the protocol messages. The new component developed for VIMS is the *Remote Attestation Module*, a module that runs on the A-VM and that implements the start-up attestation and replies to attestation requests.

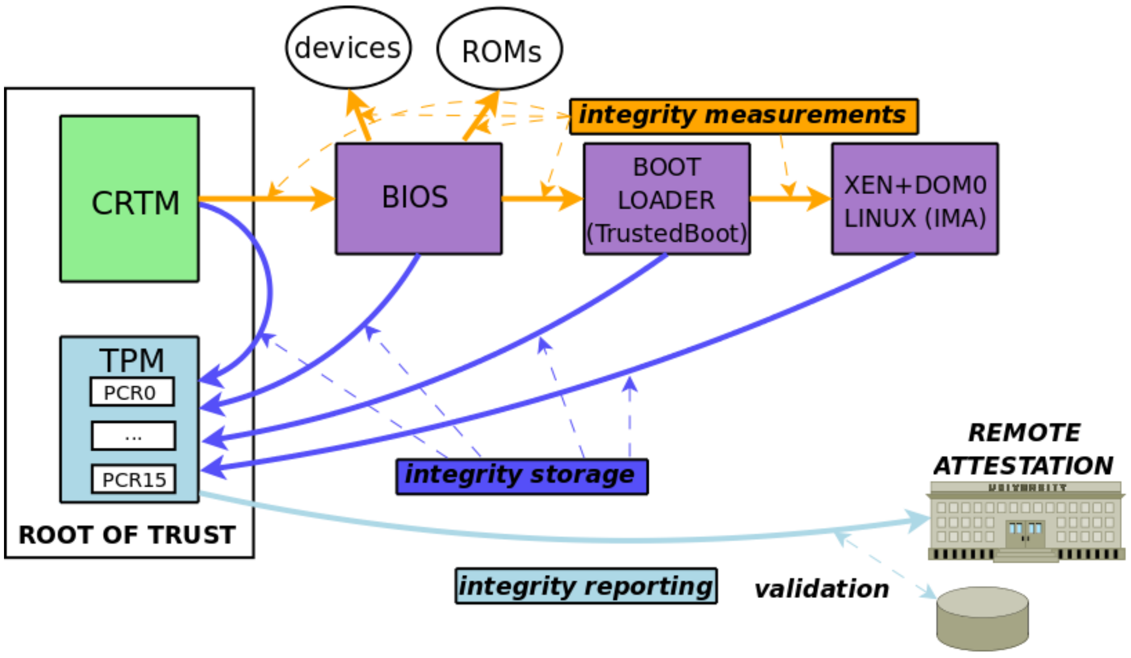


Figure 6.1: Standard Boot-Time Integrity Checks

VIMS exploits the TPM and vTPM [29] to apply the consistency checks on a Mon-VM starting from a valid root-of-trust, which is located in the hardware. However, with respect to systems that are based upon TPM mechanisms only, VIMS can implement a semantic attestation that applies consistency checks based upon the behavior of the processes. To this end, the A-VM adopts PsycoTrace strategy to protect the integrity of the kernel and of the overlay application on the Mon-VM. In this way, VIMS can: (i) guarantee the integrity of critical kernel data structures; (ii) assure that the overlay application invokes only a predefined set of system calls. This corresponds to the implementation of a *semantic integrity attestation* that applies rigorous and granular semantic checks that are strictly more powerful than those based upon hashes of running code only. In fact, by monitoring the current behavior of the Mon-VM, the A-VM applies not only the static checks implemented at boot-time that exploits the TPM (see Fig. 6.1) to verify, as an example, the integrity of the binaries of the applications that have been loaded in the Mon-VM OS, but it also monitors the semantic integrity of these applications.

The A-VM can apply alternative security policies, which can be parametrized according to:

- the frequency of the execution of security checks;
- their granularity, i.e. which data structures and software code are checked for integrity;
- the PsycoTrace strategies to describe a process self that is adopted.

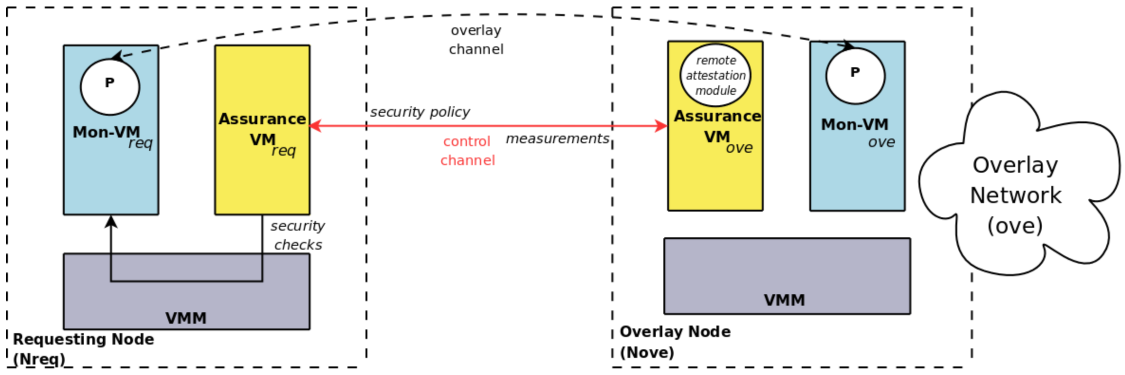


Figure 6.2: Example Scenario

Example. In the general case (see Fig. 6.2), an overlay application P running on Mon-VM$_{req}$ on a node N$_{req}$ that wants to join an overlay *ove* contacts the Remote Attestation Module of an A-VM$_{ove}$ of a node N$_{ove}$ of *ove* to connect to *ove*. Before allowing P, which is the overlay application, to connect to *ove* and access any of its services, the Remote Attestation Module on A-VM$_{ove}$ establishes an out-of-band control channel with A-VM$_{req}$. A-VM$_{req}$ measures the current configuration of P and communicates the results to A-VM$_{ove}$. Then, as long as Mon-VM$_{req}$ belongs to *ove*, A-VM$_{req}$ monitors the behavior of Mon-VM$_{req}$ and of P and it exchanges information with the Remote Attestation Module on A-VM$_{ove}$ through the control channel about (i) the integrity policy, i.e. the checks to be applied, (ii) the results of the measurements on Mon-VM$_{req}$ and P. The protocol exploits the control channel also to alert the Remote Attestation Module on A-VM$_{ove}$ anytime Mon-VM$_{req}$ or P have been compromised.

6.1.1 Formal Model

A measure of the integrity of an entity is a way to establish trust in the entity and is the building block to create a chain of trust among components. This section briefly describes a formal model to build the chains of trust from the trust relations among system components, i.e. Mon-VMs, A-VMs and TPM. The first step to describe the model is the definition of a partial order among the policies to measure the integrity of a component. This order is defined in terms of the ordering among the checks that a policy applies, e.g. among functions that map memory values into a boolean. A check C_1 is more severe than C_2 if there is at least one input where C_1 returns true while C_2 returns false while the inverse never occurs. A set S_1 is more severe than a set S_2 if, for any check in S_2, S_1 includes at least one more severe check. Obviously, both the order among sets and the one among checks are partial orders. A policy is a pair including a set of checks and an application frequency. A policy P_1 is more severe than a policy P_2 if either P_1 applies a more severe check than P_2 with at least the same frequency or if P_1 applies the same checks of P_2 with a higher frequency. An attestation that adopts a policy more severe than another one results in higher security levels of the components.

In the formal model, each component c defines:

a) $\mathit{trusted}(c)$, a set of components that c trusts to apply a policy to other components;

b) the policy to be applied to a component that does not belong to $\mathit{trusted}(c)$ so that c can trust this component;

c) for each other component d, the policy that c can apply to d. If c cannot check d, the policy includes an empty set of checks and the application frequency is set to infinite;

d) $\mathit{invoke}(c, p)$, a set of components that trust c to apply a policy p to, i.e. to check, other components;

e) the policy to be applied to a component d not belonging to $\mathit{invoke}(c)$ so that d can invoke c to apply a policy.

For any component c, $\mathit{trusted}(c)$ and $\mathit{invoke}(c, p)$ are the output of a function whose input is the physical architecture and the mapping of components onto the architecture. As an example, a VM may only trust the VMs that run on a node that includes a TPM or a VM can check another one only if they are mapped onto the same node. The set of components that c may trust is the fixed point of the equations that relates the ones that c trusts and those that c can invoke to apply a policy at least as severe as the one that it requires to trust a component. As an example, if a trusts b to apply a security policy, i.e. b belongs to $\mathit{trusted}(a)$, and b can apply to c a security policy at least as severe as the one that a requires to trust a component, then a can trust c. All the chains of trust between two components may be computed in an automatic way by automating the computation of the fixed point. Alternative models are defined according to the components that require the application of a policy. As an example, a component a may require that b applies a policy to a itself to prove to c that a can be trusted. Alternatively, c may delegate the application of a policy to b to check whether it may trust a. More general models consider the communication of policies and of the measurement results as well so that a trusts b only if, besides others, the confidentiality and/or the integrity of communications between a and b are guaranteed by components that a trusts. As far as concerns VIMS, it is worth noticing that sometimes the frequency of some checks cannot be freely chosen because some checks can be executed only when a system call is invoked, e.g. checks on the sequence of system calls issued by the overlay application.

6.1.2 Assurance Virtual Machine

VIMS attestation is *fully transparent* because, as discussed in the previous chapters, Mon-VM runs off-the-shelf software that is not aware of the integrity measurements. To measure the integrity of the Mon-VM, at first the TrustedBoot [223] is loaded and the TPM applies a set of measurements on the boot-loader, so that from now on all the steps can be measured, from boot to kernel loading and modules. Attestation requires the certification of both the A-VM on a node and of the measurements that it implements on the Mon-VM, so that the A-VMs on other nodes of the overlay can establish trust on the node integrity based upon these measurements. This is achieved by signing the hash of the running software with the TPM private key to create a chain-of-trust, from the BIOS up to the A-VM, that certifies the integrity of the VMM and of A-VM.

To this end, VIMS exploits the features of the TPM to build a hash chain on the client system that measures a predefined sequence of code loads, such as the authenticated boot of the VMM and of the kernel of the A-VM from the BIOS and boot-loader. The TPM measurements indicate that the VMM is safely started, so that it can initialize the local A-VM to assure it is started in a safe state. The A-VM in the local node can retrieve these hash values, which are called *measurements*, and protects them so that they cannot be accessed by the Mon-VM on its node. In this way, the A-VM of a node can establish trust at first into the measurements applied by the A-VM in another node and then into the run-time properties of the remote Mon-VM. The A-VM uses the message returned by the TPM quote operation to send an authenticated hash chain to another A-VM to validate the integrity of the code in the hash chain that belongs to the Mon-VM.

VIMS verifies the initial integrity of both Mon-VM and A-VM by measuring their configurations through hash functions of the code of running processes and of the Mon-VM kernel, i.e. critical data structures, code and kernel modules. After the A-VM has been safely initialized, it applies PsycoTrace run-time tools to check the integrity of the software on the Mon-VM. As an example, the A-VM may periodically retrieve the list of the kernel modules to verify that they are authorized kernel modules and to measure their integrity. A policy can also apply a description of the process self that defines invariants on variable values or on the sequence of system calls of the overlay application. Anytime the computed hash differs from the stored one or the behavior of the overlay application differs from the expected one, the Mon-VM cannot be trusted and A-VM tears down the connection.

Since PsycoTrace static tools compute a behavior that over-approximates the overlay application's run-time behavior no false positives can occur. An A-VM database stores the PsycoTrace policies it can apply on request by A-VM. Each one results in a measurement granularity and in an assurance level.

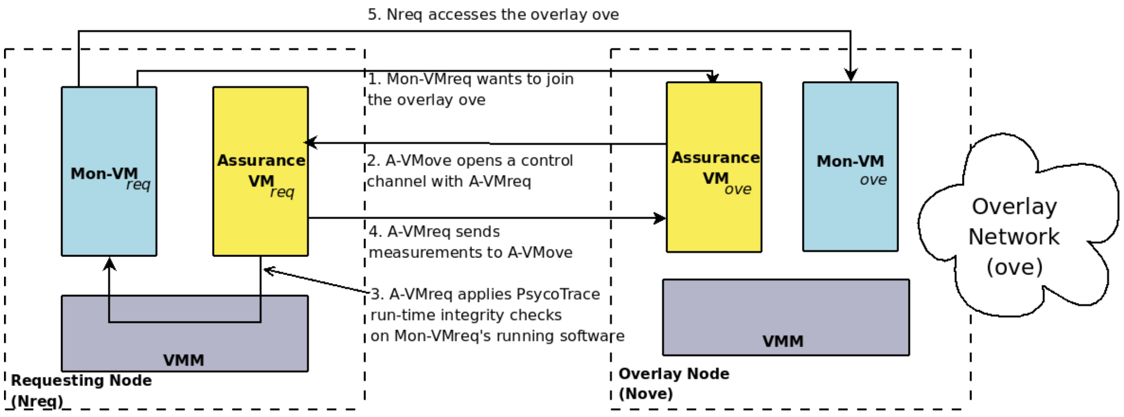

5. Nreq accesses the overlay ove
1. Mon-VMreq wants to join the overlay ove
2. A-VMove opens a control channel with A-VMreq
4. A-VMreq sends measurements to A-VMove
3. A-VMreq applies PsycoTrace run-time integrity checks on Mon-VMreq's running software
Mon-VM req
Assurance VM req
VMM
Requesting Node (Nreq)
Assurance VM ove
Mon-VM ove
VMM
Overlay Node (Nove)
Overlay Network (ove)

Figure 6.3: Start-up Attestation

6.1.3 Start-up Attestation and Monitoring

When a node N$_{req}$ tries to join an overlay *ove* (see Fig. 6.3), A-VM$_{ove}$ acts as an *appraiser* [53] that implements, on behalf of *ove*, the start-up attestation of N$_{req}$ before it can join the overlay. A-VM$_{ove}$ intercepts the request from Mon-VM$_{req}$ and it deduces the IP address of A-VM$_{req}$ from the one of Mon-VM$_{req}$.

By directly accessing and examining the run-time status of Mon-VM$_{req}$ through virtual machine introspection, the run-time tools on A-VM$_{req}$ can apply any measurements according to the appraiser's requests. The appraiser always applies hashing assertions to check the integrity of A-VM$_{req}$. After attesting the integrity of A-VM$_{req}$, A-VM$_{ove}$ can trust this VM and delegate to it the attestation of Mon-VM$_{req}$. A database in A-VM$_{req}$ records the measurements that the appraiser can request in the start-up attestation. If the attestation confirms that the configuration of Mon-VM$_{req}$ is correct, A-VM$_{req}$ starts the continuous monitoring by applying the strategy requested by the appraiser A-VM$_{ove}$.

To describe in more detail the start-up attestation, we consider the steps of the communication protocol among the various VMs (see Fig. 6.3):

1. Mon-VM$_{req}$ contacts Mon-VM$_{ove}$ to join *ove*;
2. A-VM$_{ove}$ intercepts the request and transmits to A-VM$_{req}$ the set of measurements to be applied for the start-up attestation.
3. the run-time tools on A-VM$_{req}$ compute the requested measurements;
4. A-VM$_{req}$ returns the measurements to A-VM$_{ove}$;
5. if the attestation is successful, A-VM$_{ove}$ communicates to A-VM$_{req}$ the measurements that the run-time monitoring should apply and enables Mon-VM$_{req}$ to join *ove*.

Further features of the protocol are:

- the control channel between the A-VMs exists as long as N$_{req}$ belongs to *ove*;
- A-VM$_{req}$ can apply consistency checks on Mon-VM$_{req}$ on request from A-VM$_{ove}$;
- A-VM$_{ove}$ can request that specific measurements are applied after a timeout.

6.1.4 Trust in the Measurements and in the Node Configuration

A critical issue of any attestation is the root-of-trust of the measurement system. VIMS exploits the TPM to measure the integrity of the VMs through a hash chain that measures the authenticated boot of the VMM and of the kernel of A-VM from the BIOS and boot-loader. If the VMM is safely started, then it can initialize the local A-VM to assure that also the A-VM is started in a safe state. A-VM$_{req}$ enables an appraiser A-VM$_{ove}$ to retrieve the hash values that A-VM$_{req}$ protects from accesses by Mon-VM$_{req}$. To validate the integrity of the code in the node hash chain, A-VM$_{req}$ sends to the appraiser the authenticated hash chain returned by a TPM *quote* operation. In this way, the appraiser and the overlay can establish trust at first into A-VM$_{req}$ and its measurements and then into the applications on Mon-VM$_{req}$.

After its safe initialization, A-VM$_{req}$ can continuously apply measurements to monitor the integrity of Mon-VM$_{req}$ and of P but this is ineffective against modifications of the configuration of N$_{req}$. To this purpose, VIMS introduces a *seal-plugin* module that exploits the TPM functions *seal* and *unseal*. During the start-up attestation of N$_{req}$, as soon as the appraiser A-VM$_{ove}$ has attested the integrity of A-VM$_{req}$, it asks this A-VM to seal a value K by listing all the TPM registers. From now on, the appraiser can control the integrity of the configurations of N$_{req}$ by a challenge/response protocol that sends a nonce n encrypted with K. A-VM$_{req}$ has to unseal K, decrypt the received value and return n to the appraiser. Obviously, A-VM$_{req}$ can reply to the challenge only if the configuration of N$_{req}$ is stable. To avoid the loss of K when A-VM$_{ove}$ leaves the overlay, K can be distributed to other A-VMs both at the end of the start-up attestation and before N$_{ove}$ leaves the overlay. In this way, each A-VM holds a set of keys, each for a distinct A-VM, and it may randomly choose one of them to control the configuration of the corresponding node. Two consecutive controls are separated by an interval T that depends upon the overlay security policy. The handling of the seal-plugin module shows how A-VMs can cooperate to create an overlay-wide appraiser.

The seal-plugin module may be ineffective against *intermittent configuration attacks* where a node switches from a malicious configuration to a correct one, and the other way around, with a frequency that depends upon $1/T$ so that its configuration is correct anytime it is controlled. To detect these attacks, the time in-between two consecutive controls may be randomly chosen from a distribution with an average equal to T. However, since the configuration of an A-VM may be controlled by several other ones and the clocks of these A-VMs are not tightly synchronized, the complexity of foreseeing the timing of configuration controls is rather high even if the time in-between two controls is fixed.

Finally, if the node includes a TPM, the measurements can be trusted provided that the configurations of the VMM and of the A-VM are trusted and that the seal-plugin module can prevent configuration updates. This is often the case if the nodes are in controlled environment, such as a corporate network. If physical attacks against the memory of a node, or other hardware components, cannot be avoided, then increasing the security of the overlay is very hard. However, robustness with respect to physical attacks can be improved by adopting code obfuscation techniques described in Chap. 7.

6.1.5 Handling of Anomalous Behavior

Anytime an A-VM discovers that the attestation of a Mon-VM fails or that the behavior of P differs from the expected one, it kills the Mon-VM. The A-VM may also communicate this decision to the A-VMs of some overlay nodes connected to the considered one so that they update a blacklist of IP addresses that cannot belong to the overlay. We recall that the continuous monitoring cannot produce a false positives because the description computed by the static tools over-approximates the run-time behavior of P. Instead, if an A-VM detects that another A-VM cannot be attested or the configuration of the node has been updated, it communicates this information to other A-VMs that can either inform the Mon-VM on their nodes or simply destroy the connection to the removed node. It is useless to inform the A-VM of the disconnected node because it is untrusted. Notice that, to prevent denial of service attacks, before disconnecting the node, the integrity of the A-VM requiring the disconnection should be attested.

6.2 Current Implementation

Xen [26] 3.1.0 is the adopted technology to create the VMs, which are based on Debian Etch 4.0 with Linux kernel 2.6.18. We adopted PsycoTrace tools described in Chapter 4 and 5 to define the semantic checks on the overlay application and PsycoTrace Introspection Library to compute the assertion on the Mon-VM memory. The various modules that run on A-VM that have been implemented are (see Fig. 6.4):

- Remote Attestation Module: an A-VM module that implements the start-up attestation and implements the attestation protocol;
- a database with the measurements that can be applied;

- a vTPM module interface;
- an interface for the low-level introspection function;
- an OpenVPN plugin [173] used to connect the node to a VPN;
- an extension to a Gnutella code [222].

The last two modules act as interfaces to join, respectively, a VPN and a Gnutella network. The attestation protocol has been implemented in Java. The attestation library consists of 4 Java classes of about 1500 lines of code, whereas the monitoring requires 1000 lines of Java code, including a small wrapper to interface with the introspection functions. The update of the Gnutella code to support the attestation consists of about 1500 lines of code.

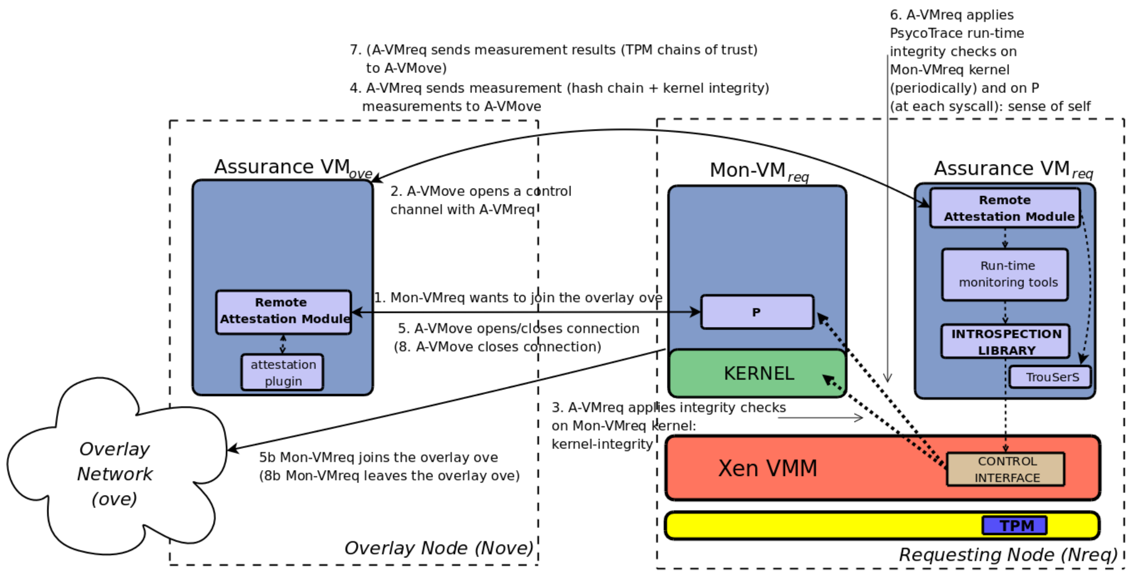

6. A-VMreq applies PsycoTrace run-time integrity checks on Mon-VMreq kernel (periodically) and on P (at each syscall): sense of self
7. (A-VMreq sends measurement results (TPM chains of trust) to A-VMove)
4. A-VMreq sends measurement (hash chain + kernel integrity) measurements to A-VMove
Assurance VM$_{ove}$
2. A-VMove opens a control channel with A-VMreq
Mon-VM$_{req}$
Assurance VM$_{req}$
Remote Attestation Module
Run-time monitoring tools
INTROSPECTION LIBRARY
TrouSerS
1. Mon-VMreq wants to join the overlay ove
Remote Attestation Module
P
5. A-VMove opens/closes connection
(8. A-VMove closes connection)
attestation plugin
KERNEL
3. A-VMreq applies integrity checks on Mon-VMreq kernel: kernel-integrity
Overlay Network (ove)
5b Mon-VMreq joins the overlay ove
(8b Mon-VMreq leaves the overlay ove)
Xen VMM
CONTROL INTERFACE
TPM
Overlay Node (Nove)
Requesting Node (Nreq)

Figure 6.4: Current Implementation

In the first experiments, the Mon-VM runs a VPN client to join a remote Intranet and Remote Attestation Module acts as a VPN server on N$_{ove}$ (see Fig. 6.5). This covers those cases where an Intranet hosts critical resources such as SCADA devices that are remotely accessed and monitored. To protect the integrity of the VPN client, the continuous monitoring strategy evaluates assertions computed by the static tools. The VPN server has been modified to handle the remote attestation protocol. Java SSL libraries and TPM/J [208] are used to access the TPM values and to create a VPN connection. Finally, OpenVPN has been extended with plugins to enable remote attestation. In the second experiment (see Fig. 6.6), we have extended a Gnutella node to support attestation. In both experiments, we assume that no measurement violates the user's privacy.

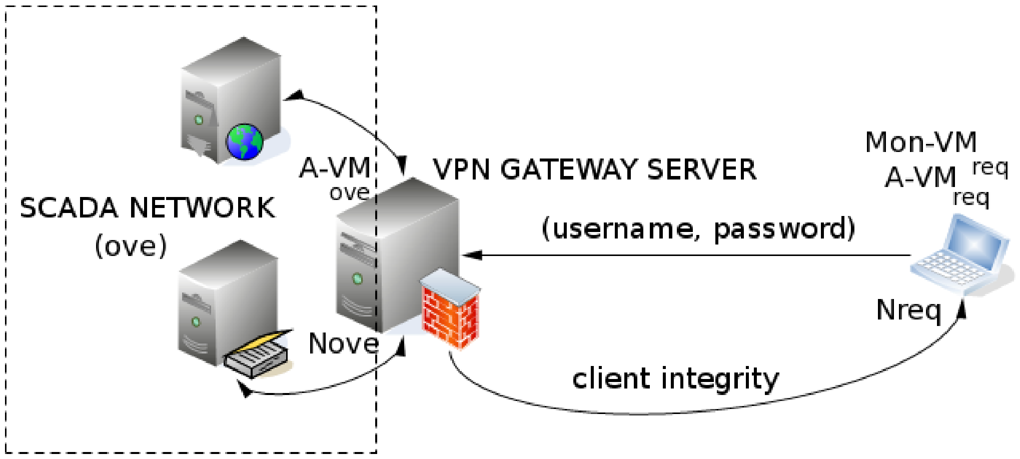


Figure 6.5: First Testbed Implementation

6.2.1 Remote Attestation Module

The Remote Attestation Module is at the core of the attestation protocol as it initializes the protocol and implements the communications between N_{ove}, a node of the overlay, and A-VM_{req}, the A-VM on the node that is trying to connect to the overlay. The protocol is triggered each time Mon-VM_{req} tries to join an overlay by opening a connection to N_{ove}. The Remote Attestation Module on A-VM_{ove} acts as daemon service waiting for connections and it starts the handshaking phase. In the VPN case, this module is an OpenVPN plugin activated by requests to access the private network. Instead, in the P2P testbed scenario, the module on the A-VM cooperates with a new thread in the Gnutella code to manage the control channel between the A-VMs. Once activated, the Remote Attestation Module maps the IP address of Mon-VM_{req} into the one of A-VM_{req} and it opens a connection to the A-VM_{req}.

The handshaking between the Remote Attestation Module on A-VM_{ove} and A-VM_{req} includes the mutual authentication and the initial parameters exchange. At the end of the handshake, A-VM_{req} applies the initial set of measurements to Mon-VM_{req} and it transmits their results and the hashes computed by the TrustedBoot to the Remote Attestation Module on A-VM_{ove}. To attest the integrity of A-VM_{req} and discover the current configuration of Mon-VM_{req}, the Remote Attestation Module on A-VM_{ove} compares the hashes against those in the database. Then, A-VM_{ove} defines a security level and it communicates this level to A-VM_{req}. Finally, A-VM_{req} stores this level in its database and it applies the corresponding strategy. As an example,

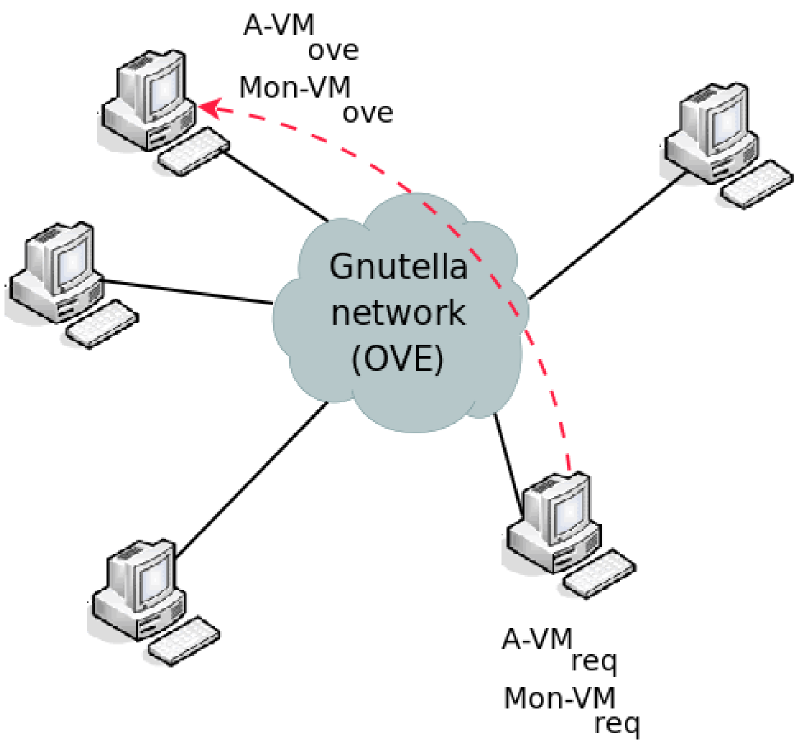


Figure 6.6: Second Testbed Implementation

a node is attested when it joins the overlay and anytime a predefined number of communications has occurred.

Currently, the Remote Attestation Module can apply further measurements besides those that implement the strategies previously described. In fact, the Remote Attestation Module may apply either an *on-demand* or a *frequency* policy, based on a XML-encoded policy (see Tab. 6.1 for an example of an attestation response from the A-VM$_{req}$). In an on-demand policy, as long as a node belongs to an overlay, the Remote Attestation Module may request specific measurements to an A-VM. These policies include all the strategies previously described to detect an attack against the overlay application. Instead, in frequency-based policies, the A-VM computes the hash values of some memory areas, as requested by the Remote Attestation Module on the appraiser A-VM, with the corresponding frequency and return these values to the module.

6.2.2 Description of the Attestation Protocol

The protocol between A-VM$_{req}$ and the Remote Attestation Module on A-VM$_{ove}$ includes the following steps (see Fig. 6.7):

1. Mon-VM$_{req}$ opens a connection to the Remote Attestation Module on A-VM$_{ove}$;

```
<attestation>
  <response>
    <overallresult>suspicious</overallresult>
    <nonceid>89892345</nonceid>
    <AssuranceIp>192.168.1.1</AssuranceIp>
    <reqpolicy>timeout</reqpolicy>
    <timemillis>20000</timemillis>
    <assurancelevel>4</assurancelevel>
    <measuresaggregate>
      <measureschain>
        <measure>
          <idmeasure>TPM-Boot</idmeasure>
            <hash>7844e409c39fad83bb65f7dac4c8a53e</hash>
            <status>trusted</status>
            <object>TPM.Measure</object>
            <name>boot</name>
        </measure>
        <measure>
          <idmeasure>0</idmeasure>
            <hash>7844e409c39fad83bb65f7dac4c8a53e</hash>
            <status>trusted</status>
            <object>kernel.struct</object>
            <name>idt_table</name>
          </measure>
           .................
        <measure>
          <idmeasure>78</idmeasure>
            <hash>0000e409c39fad83bb65f7dac4c8a53e</hash>
            <status>trusted</status>
            <statusprofile>wrong</statusprofile>
            <object>process</object>
            <name>modprobe</name>
            </measure>
        </measureschain>
      </measuresaggregate>
  </response>
</attestation>
```

Table 6.1: Example of Attestation Response

2. the Remote Attestation Module on A-VM$_{ove}$ checks the user's credentials;

3. if Mon-VM$_{req}$ has been authenticated, the Remote Attestation Module on A-VM$_{ove}$ invokes a function to query the database;

4. the Remote Attestation Module on A-VM$_{ove}$ retrieves the IP address of A-VM$_{req}$ either though a function or by mapping Mon-VM$_{req}$ address;

5. the Remote Attestation Module on A-VM$_{ove}$ sets up a control channel with A-VM$_{req}$;

6. the Remote Attestation Module on A-VM$_{ove}$ sends the handshake message with some parameters;

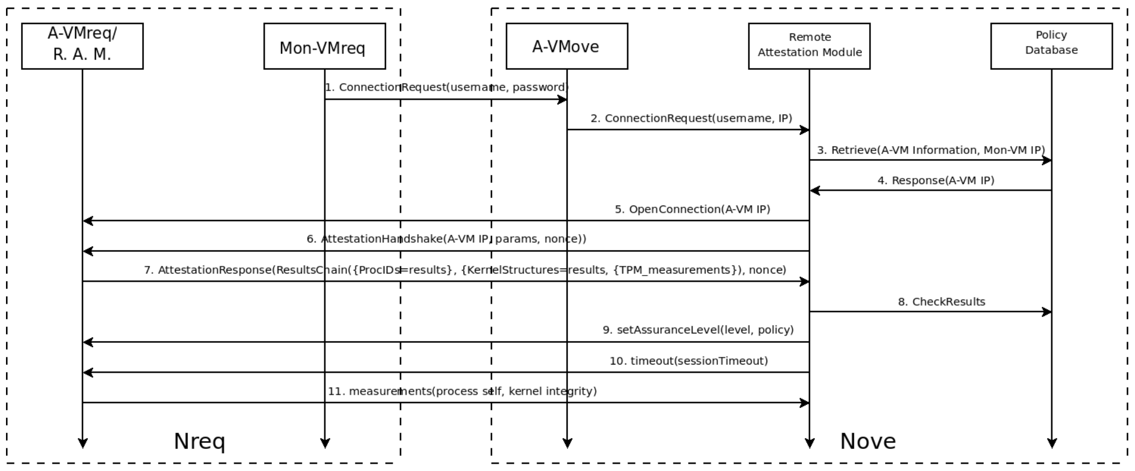

A-VMreq/
R. A. M.
Mon-VMreq
A-VMove
Remote
Attestation Module
Policy
Database
1. ConnectionRequest(username, password)
2. ConnectionRequest(username, IP)
3. Retrieve(A-VM Information, Mon-VM IP)
4. Response(A-VM IP)
5. OpenConnection(A-VM IP)
6. AttestationHandshake(A-VM IP, params, nonce))
7. AttestationResponse(ResultsChain({ProcIDs=results}, {KernelStructures=results, {TPM_measurements}), nonce)
8. CheckResults
9. setAssuranceLevel(level, policy)
10. timeout(sessionTimeout)
11. measurements(process self, kernel integrity)
Nreq
Nove

Figure 6.7: Attestation Protocol Overview

7. A-VM$_{req}$ computes the hash values of P and sends them to the Remote Attestation Module on A-VM$_{ove}$ chained with the TPM's measurements of the underlying system;

8. the Remote Attestation Module on A-VM$_{ove}$ compares the results sent by A-VM$_{req}$ against the expected ones;

9. the Remote Attestation Module on A-VM$_{ove}$ sends to A-VM$_{req}$ an XML configuration file with the measurements that it should apply. The function parameters are (i) `level`, an ID coupled with a strategy in A-VM$_{req}$ database; (ii) `measurement`, whose value can either be `frequency` or `on demand`;

10. in a `frequency` policy, the Remote Attestation Module on A-VM$_{ove}$ sends a timeout value for the session. A-VM$_{req}$ applies the measurements when the timeout is elapsed and sends the results to the Remote Attestation Module on A-VM$_{ove}$;

11. from this moment on, A-VM$_{req}$ measures the integrity of the overlay application and of the kernel and it returns the measurements to the Remote Attestation Module on A-VM$_{ove}$.

6.2.3 Measurements in a P2P Overlay

Our solution to protect a P2P overlay through VIMS is not fully transparent because, to compute the integrity measurements, we have modified three distinct components in the Gnutella code [222] that implement, respectively, the handshake, the download and the exchange of Ping messages. Moreover, a further thread in the Gnutella code implements the message exchange between the appraiser A-VM and P (i.e., the Gnutella application). As an example, if the new thread receives an alert from an A-VM stating that a node has been compromised, it forces P to "kick" the compromised node out of the overlay. Anytime a node attempts to connect to a Gnutella peer, the new thread receives the request and informs the A-VM on its node to act as an appraiser and to interact with the A-VM on the requesting node. As soon as this node has been successfully attested, it can join the overlay. From now on the standard Gnutella protocol is executed. If required, also the the requesting node can attest the Gnutella peer.

The handshake component has been modified to implement a revised version of the original protocol where (see Fig. 6.8):

1. Mon-VM$_{req}$ opens a TCP connections with Mon-VM$_{ove}$;
2. Mon-VM$_{req}$ sends the string "GNUTELLA CONNECT/0.6" to Mon-VM$_{ove}$;
3. Mon-VM$_{req}$ sends a header with its specifications containing new fields for the attestation to Mon-VM$_{ove}$;
4. Mon-VM$_{ove}$ contacts A-VM$_{ove}$ to inform that Mon-VM$_{req}$ is willing to join the Gnutella Overlay;
5. A-VM$_{ove}$ contacts A-VM$_{req}$ using UDP to request the integrity measurements for Mon-VM$_{req}$;
6. A-VM$_{req}$ applies the initial integrity measurements, i.e. the hashing measurements on Mon-VM$_{req}$, and sends the results to A-VM$_{ove}$;
7. A-VM$_{ove}$ communicates the response to Mon-VM$_{ove}$;
8. if the response is positive, Mon-VM$_{ove}$ sends the string "GNUTELLA/0.6 200 OK" to Mon-VM$_{req}$, otherwise it closes the connections.

Then, steps 4 to 8 are repeated but this time the roles of Mon-VM$_{ove}$ and Mon-VM$_{req}$ are reversed, i.e. Mon-VM$_{req}$ requires A-VM$_{ove}$ to attest Mon-VM$_{ove}$.

The following is an example of the handshake messages exchanged between two nodes:

```
1)
GNUTELLA CONNECT/0.6
Node: 123.123.123.123:1234
Pong-Caching: 0.1
GGEP: 0.5
Ip-Att: 123.123.123.123
Port-Att: 6666
AttEveryXPing: 5
Dom-Name: Mon-VMreq
2)
```

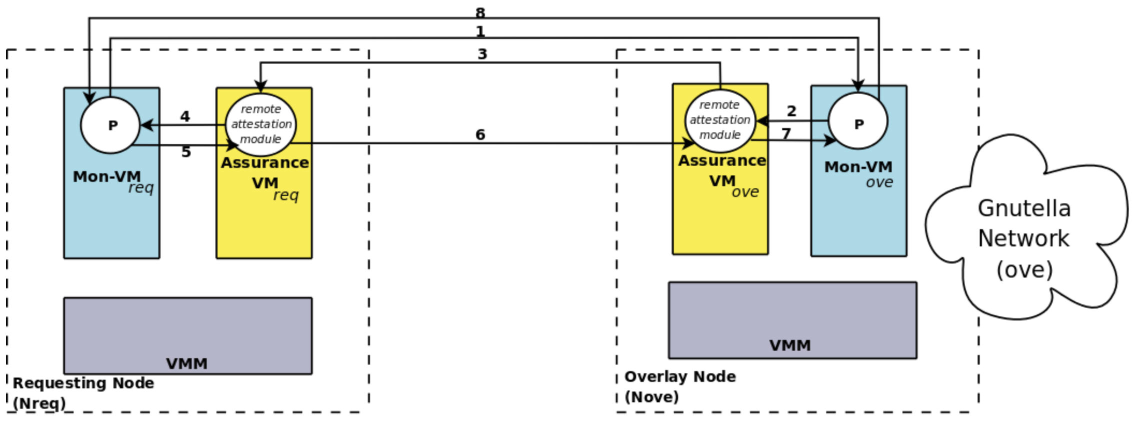


Figure 6.8: Protocol in the Gnutella Implementation

```
ATTESTATION MESSAGE
Code-Message: ATTESTATION_ON_HANDSHAKE
Ip-Node: 123.123.123.123
Port-Node: 1234
Ip-Att: 123.123.123.123
Port-Att: 6666
AttEveryXPing: 5
Dom-Name: Mon-VMreq
3)
ATTESTATION MESSAGE
Code-Message: REQUEST_ATTESTATION
Dom-Name:  Mon-VMreq
4-5)
Integrity measurements on Mon-VMreq
6)
ATTESTATION MESSAGE
Code-Message: ATTESTATION_ON_HANDSHAKE
Attestation: OK
7)
ATTESTATION MESSAGE
Code-Message: ATTESTATION_ON_HANDSHAKE
Attestation: OK
8)
GNUTELLA/0.6 200 OK
User-Agent: gtk-gnutella/0.95
Pong-Caching: 0.1ss
GGEP: 0.5
Ip-Att: 234.234.234.234
Port-Att: 8888
AttEveryXPing: 5
```

```
Dom-Name: Bar
Private-Data: 5ef89a
```

As soon as Mon-VM receives the "OK" string, steps 2-8 are repeated to attest the integrity of Mon-VM$_{ove}$. We do not detail here the update of the components implementing the downloads and Ping messages, since it recalls that of the handshake. It suffices to say that the original protocol has been modified to include integrity measurements as well. Anytime a file download is started, the A-VMs of the nodes involved in the download cooperate to attest the integrity of both Mon-VMs. Attestations may be fired by Ping messages as well. As soon as Mon-VM receives a Ping message from another Mon-VM it replies with a Pong and then, if the number of Ping messages exceeds a threshold, it starts the mutual attestation of both Mon-VMs. As soon as the attestation of a peer Mon-VM fails, the appraiser A-VM informs the local Mon-VM and all its neighbor A-VMs that, in turn, inform the Gnutella application on the corresponding Mon-VMs to close the connection with the kicked Mon-VM. Furthermore, since A-VM continuously monitors the behavior of P, Ping messages also include integrity results based upon measurements on P and based upon hashing assertions.

6.3 Performance Results

In the following, we discuss at first the overhead of start-up attestation and then the one of continuous monitoring. The system to run the prototype included a Pentium Centrino Duo T2250 1.7GHz. In all the tests, the A-VM Linux kernel version was 2.6.18-xen, 128MB of physical memory were allocated to the Mon-VM, running a Linux Debian distribution, and 874 MB to the A-VM.

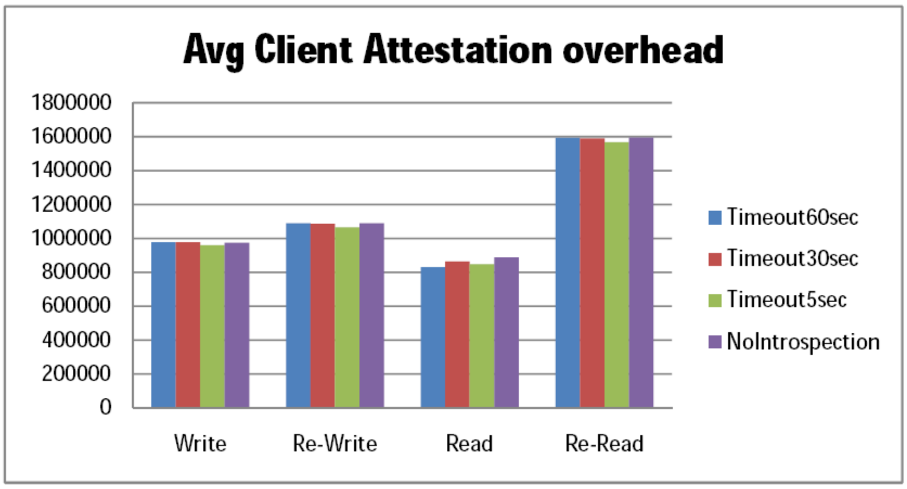


Figure 6.9: Average Client Attestation Overhead

6.3.1 Attestation

Figure 6.9 shows the number of operations of the IOzone [120] benchmark on the Mon-VM when the attestation is repeated after an interval from 5 sec to one minute. In this case, network latency masks the protocol overhead.

Figure 6.10 shows the attestation overhead computed by IOzone on the Mon-VM when 3 nodes try to connect to the overlay simultaneously. The figure shows the overhead as a function of timeout, the time interval in-between two consecutive attestations.

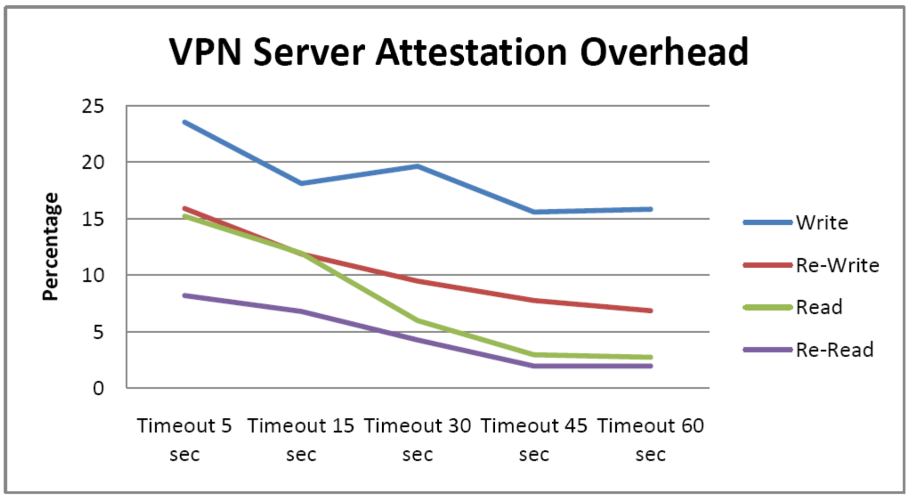


Figure 6.10: VPN Server Attestation Overhead

To evaluate the overhead of the attestations of the integrity of the Mon-VM kernel and of applicative software in a Gnutella overlay, we have developed a solution that is not fully transparent because a further thread in the Gnutella application intercepts connection requests. A node is attested anytime it joins a network, downloads a file or has exchanged a policy defined number of ping messages. Anytime a node attempts to connect to a Gnutella peer, the newly inserted Gnutella thread in the peer receives the message and informs the A-VM on its node to act as an appraiser and to interact with the A-VM on the requesting node. As soon as this node has been successfully attested, it can join the overlay. From now on the standard Gnutella protocol is executed.

The attestation overhead is low because the increase in the execution time is less than 10 percent if the Mon-VM only runs the Gnutella application. If it runs other applications, the slow-down of the download due to the attestation may be neglected provided that less than one attestation for minute occurs. As far as concerns the communication overhead, the number of exchanged messages is increased from four in the original protocol to, at most, eight to implement the attestation and, however, two of these messages are exchanged between VMs on the same node.

Chapter 7

Code Obfuscation in a Virtual Environment

Code obfuscation is the practice of making code unintelligible to prevent its reverse engineering by applying a set of transformations that change the physical appearance of the code, while preserving its original semantics. We present an obfuscation transformation targeted at cloud environments that strongly separates the obfuscated program from the information to compute the inverse transformation. This separation is achieved by exploiting PsycoTrace run-time architecture. To this end, the Mon-VM stores the obfuscated program as a set of program fragments, whereas the I-VM stores the information to invert the obfuscation. The proposed obfuscation strategy applies the encryption of program fragments and the randomization of the program control-flow. At anytime, exactly one program fragment, the current one, is stored in clear on the Mon-VM, whereas any other fragment is encrypted. As soon as the execution of the current fragment terminates, control is transferred to the I-VM that decrypts the next fragment to be executed, encrypts the current one and updates the program counter of the Mon-VM so that it can correctly execute the next fragment. Alternative solutions may be implemented according to the definition of fragment that is adopted. In particular, we consider that a fragment includes all the instructions in-between two system calls as this minimizes the overhead to intercept the flow of control on the Mon-VM. In fact, by trapping the execution of a system call on the Mon-VM, the I-VM is alerted that a fragment has been completely executed so that it can encrypt and decrypt the proper fragments and update the program counter. This corresponds to a program decomposition into *system blocks*, where the notion of system block is similar to that of basic block provided that control transfers are replaced by system calls. A noticeable advantage of this solution is that is fully transparent to system software.

7.1 Threat Model

The threat model we assume is focused on a cloud service provider (CSP) that implements an infrastructure-as-a-service model through the instantiation of dedicated VMs where the user has full access, i.e. the VMs are *bare metal*. According to the environments, i.e. OS and development tools, the vendor may also offer a minimal set of applications to the cloud users: in this scenario, the CSP is interested in obfuscating these applications. (This model may be viewed as a special case of platform-as-a-service model where the user can also apply changes at the OS-level.) Hence, the user can access the highest privilege level of the VM so that she can read and modify any programs in memory or on disk provided by the CSP. For this reason, the CSP should run an I-VM to obfuscate the program that it provides to the users. In the VIMS scenario, if the start-up attestation is successful, the overlay sends (a) the obfuscated version of the overlay program to the Mon-VM and (b) the system block graph and the decryption keys to the A-VM.

7.2 System Blocks and Program Representation

This section briefly introduces the program representation underlying the proposed obfuscation model.

The fundamental notion of the model is that of system block. Informally, a system block is the program fragment that includes all the instructions that may be executed in-between two consecutive system calls.

Definition (System Block (SB))**.** *If s is a system call or the first instruction of the program,* a system block *(SB) is the smallest program fragment that includes any instruction that may be executed in-between s (not included) and either the next system call (included) or the program end.*

A SB can be described as a sub-graph of the program control-flow graph that has just one entry point, the instruction following s, and several exit points. There is a distinct exit point for each system call that can be executed immediately after s, or for the program end if s is the last call that may be executed before the end is reached. While a SB recalls a basic block (BB), the two notions are fully orthogonal. As a matter of fact, a BB may include several SBs, because a sequence of instructions without jumps may also include several system calls. On the other hand, a SB can include several BBs that do not issue system calls.

A program may be described as a set of SBs and a system block graph that denotes the execution order among these SBs.

Definition (System Block Graph (SBG))**.** *A* system block graph *(SBG) is an oriented graph where each node represents a distinct SB and each arc is coupled with a distinct system call among those executed by the program. An arc denotes that the*

SB represented by the arc destination node (the destination SB) is executed after the source SB, which is the SB represented by the source node. The system call coupled with the arc is the last instruction of the source SB.

In general, there may be several arcs leaving a node, because there may be several exit points for each SB, each corresponding to the execution of a distinct system call, if any.

System Blocks Example. The following code:

```
 1  read();
 2  if(x) {
 3     x = x + 1;
 4  }
 5  else{
 6     x = x + 2;
 7     write();
 8     y = y*y;
 9  }
10  z = -z;
11  time();
```

includes two SBs:

1. the first SB begins at the instruction (line 2) after the `read()` system call and ends in two points, i.e. in the `write()` (line 7) and `time()` (line 11) system calls. Hence, this SB includes the instructions `x = x + 1;`, `z = -z;`, `time();` and `x = x + 2;`, `write();`

2. the second SB begins at the instruction (line 8) after the `write()` system call, and ends in the system call `time()` (line 11). This SB contains the instructions: `y = y*y;`, `z = -z;`, `time();`

While distinct SBs may share some instructions, this sharing can be avoided by replicating shared instructions. As an example, the two SBs in the previous example share the instructions `z = -z; time();`. The sharing may be avoided by replicating the two instructions in each branch of the `if` statement.

7.2.1 Representing the Program Through System Blocks

Definition ($CFGraph(P)$)**.** *$CFGraph(P)$ is the control graph of P defined in terms of BBs.*

Definition ($SBG(P)$)**.** *$SBG(P)$ is the system block graph of P.*

$SBG(P)$, the graph that describes the decomposition of a program P into SBs, is a transformation of $CFGraph(P)$. The transformation assumes, without any loss of generality, that $CFGraph(P)$ includes exactly one initial BB and a final BB.

Currently, we build $CFGraph(P)$ from $SourceCode(P)$ but, in a real-world scenario, the graph should be deduced from the executable code where all the required libraries are statically linked through disassembly [241]. This guarantees the consistency between the statically generated $CFGraph(P)$ and the run-time behavior. The transformation is built around the notion of unit block and unit block graph, which is an intermediate representation to map $CFGraph(P)$ into $SBG(P)$ through unit blocks.

Definition (Unit Block)**.** *A* unit block *(UB) is any sequence of instructions that belong to the same BB in-between two consecutive delimiters of the BB, where a delimiter is either a system call or the first and the last instruction of the BB.*

Conceptually, a unit block generalizes a BB because also system call instructions are considered as branch instructions. If the first instruction of a UB is the first program instruction or it immediately follows a system call, then the UB is the *initial UB* of a SB, whereas UBs having as their last instruction a system call, or the last instruction of a program, are *final UBs* of the corresponding SB.

Unit Blocks Example. The following code:

```
if(x){
   x = x + 1;
   time();
   t = t + 10;
   s = s * 4;
}
else{
   y = y*y;
   m = m - 1;
   write();
   f = f + 5;
}
z = -z;
read();
```

includes five UBs, i.e.:

1. `x = x + 1; time();`
2. `t = t + 10; s = s * 4;`
3. `y = y*y; m = m - 1; write();`.
4. `f = f + 5;`.
5. `z = -z; read();`.

UB 4 is an initial UB because its first instruction immediately follows a system call, whereas UBs 3 and 5 are final UBs.

Definition (Unit Block Graph (UBG))**.** *A* unit block graph *(UBG) is a transformation of $CFGraph$ in a graph that contains a node for each UB and an arc from the node representing UB_1 to the one representing UB_2 if:*

a) UB_1 and UB_2 belong to the same BB and UB_2 is executed immediately after UB_1, or

b) UB_1 is the last UB of BB_1 and UB_2 is the first UB of BB_2 and BB_2 may be executed after BB_1.

Definition ($UBG(P)$)**.** *$UBG(P)$ is the unit block graph of P.*

In the following, we assume that system calls are represented through well-known lexical tokens, so that they can be recognized when parsing the code. As an example, if the algorithm is applied to the executable code, it should locate all `int $0x80` and `syscall` assembly instructions (i.e., those instructions to issue system calls). If, instead, it is applied to $SourceCode(P)$, these tokens represents the LIBC wrappers for system calls. The algorithm builds $SBG(P)$ by locating all the system calls inside $CFGraph(P)$ and, for each system call, it discovers all the control paths from this system call to the next system call. Then, the algorithm merges all the instructions along these paths into a new SB and it inserts the corresponding node into $SBG(P)$. The number of arcs leaving this node depends upon the number of system calls that end the block because there is a distinct arc for each system call.

In more details, the algorithm to map $CFGraph(P)$ into $SBG(P)$:

1. splits each BB containing $n > 0$ system calls into $n+1$ UBs $(1, ..., n+1)$, where the i-th block includes all the instructions in-between the $(i-1)$th system call of the BB (if $i = 1$, from the first instruction of the BB) and the i-th system call of the BB (if $i = n + 1$, to the last instruction of the BB);

2. generates $UBG(P)$;

3. visits $UBG(P)$ and, for each node n that represents a UB:

   - it starts a depth first visit of the graph;
   - determines $Succ(n)$, the set that includes any node m that represents either a UB that may be executed after n and that ends with a system call or the `END` block that terminates the program;
   - merges all the UBs represented by nodes on the path from n to any node in $Succ(n)$ into the same SB, $SB(n)$.

As an example, Fig. 7.1(b) and 7.1(c) show, respectively, the UBG and the SBG resulting from the control-flow graph shown in Fig. 7.1(a).

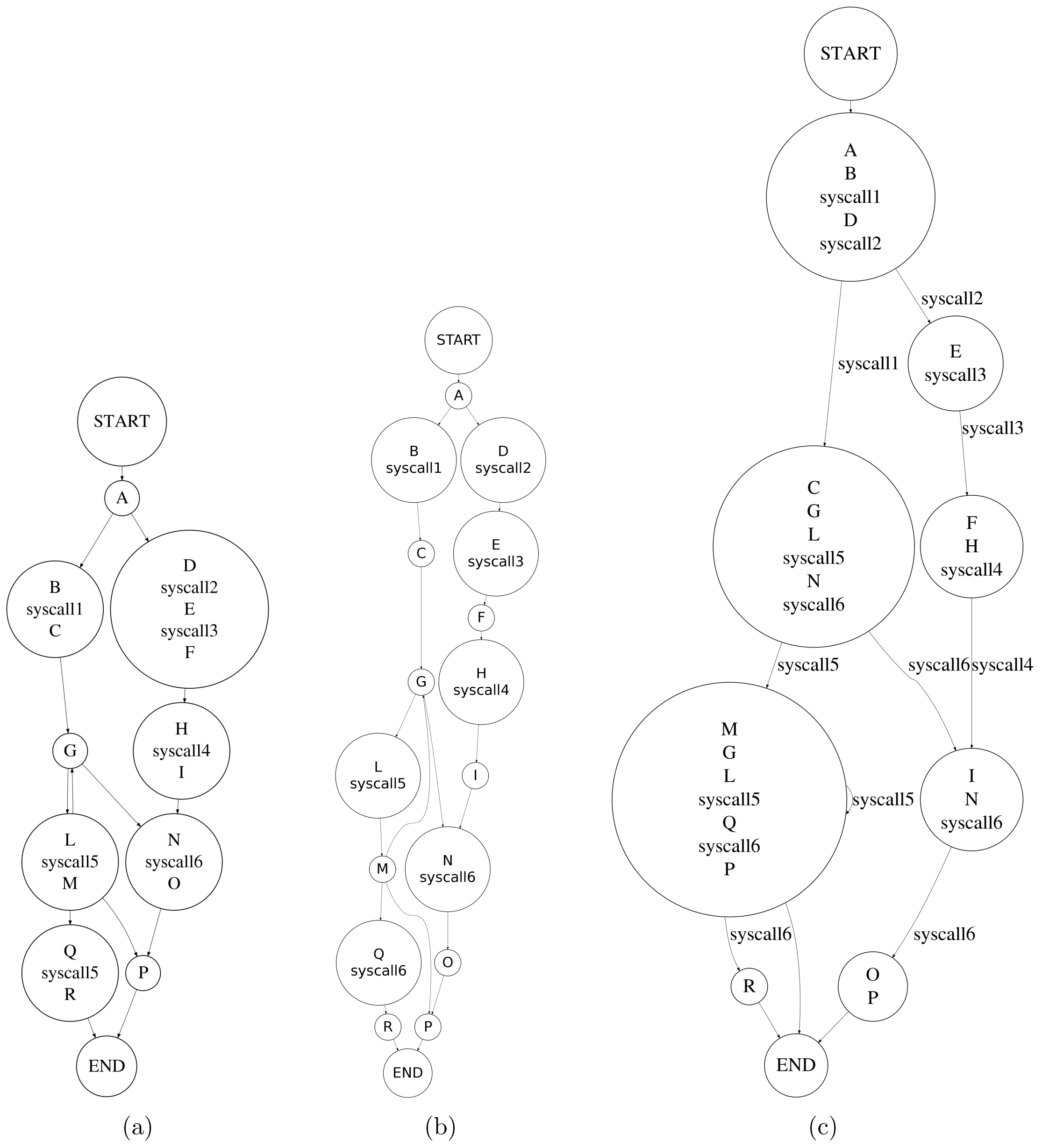


Figure 7.1: Control-Flow Graph (a) Unit Block Graph (b) System Block Graph (c)

7.2.2 Algorithm to Build the System Block Graph

In the following, we describe a high-level version of the algorithm that implements the steps previously described to locate SBs and build the SBG. Here, `SB` refers to the set of SBs that have already been located, whereas `UB` refers to the set of UBs that may follow the final UB of the `SB`. The functions are:

- `systemBlockGraphGenerator`: this is the main function that takes as parameter a reference to the basic block that is the root of the control-flow graph.

This function generates the SBG and returns a pointer to its root;

- `searchInFoundSystemBlocks`: the input of this function is a pointer to a basic block BB and returns a pointer to a SB. At first, this functions verifies if BB is an initial UB of any SB in `SB`. In this case, it returns a pointer to that SB, otherwise it returns `null`;

- `systemBlockDiscoverer`: it takes as input a basic block BB and returns the set of UBs that compose the SB having BB as its root. Moreover, it inserts into `UB` the UBs that follow the final UBs of the last SB. The search of new SBs starts from these UBs.

The following is the high level description of the algorithm:

```
SystemBlock systemBlockGraphGenerator(BasicBlock bb)
{
   SystemBlock SB = searchInFoundSBlocks(bb);
   if(SB != null)
     return SB ;
   else
   {
      SB = new SystemBlock(systemBlockDiscoverer(bb));
      addSystemBlock(SB); //add the new system block
                          //to the set of system blocks
      for each i in UB
      {
         setNextNode(SB, systemBlockGraphGenerator(i));
      }
      return SB;
   }
}

void systemBlockDiscoverer(BasicBlock bb)
{
   P = {};    // the set of unit blocks in the
              // current system block
   Q = {bb}; // set with the nodes of the control-flow graph
              // (i.e., basic blocks) to be analyzed

   repeat
      select i from Q
      Q = Q - {i};
      for each (i , j) in exit-arcs(i)
      {
         P = P + {i};
         if(isAFinalUnitBlock(i))
         {
            UB = UB + {j};
         }
         if(j is not in P )
         {
            Q = Q + {j};
         }
      }
   until(Q = {})
   return P;
}
```

7.2.2.1 Examples of System Block Graphs

In the following, we discuss three examples of SBG generation.

Example 1. Consider the following snippet of code:

```
if(x) {
   x = x + 2;
}
else{
   x = x + 3;
   read();
   y = y * y;
}
z = -z;
x = x * 2;
write();
y = y + 10;
s = s + 1;
time();
```

The algorithm explores the control-flow graph starting with the first basic block, which contains the first UB with index 1 (see Fig. 7.2(a)).

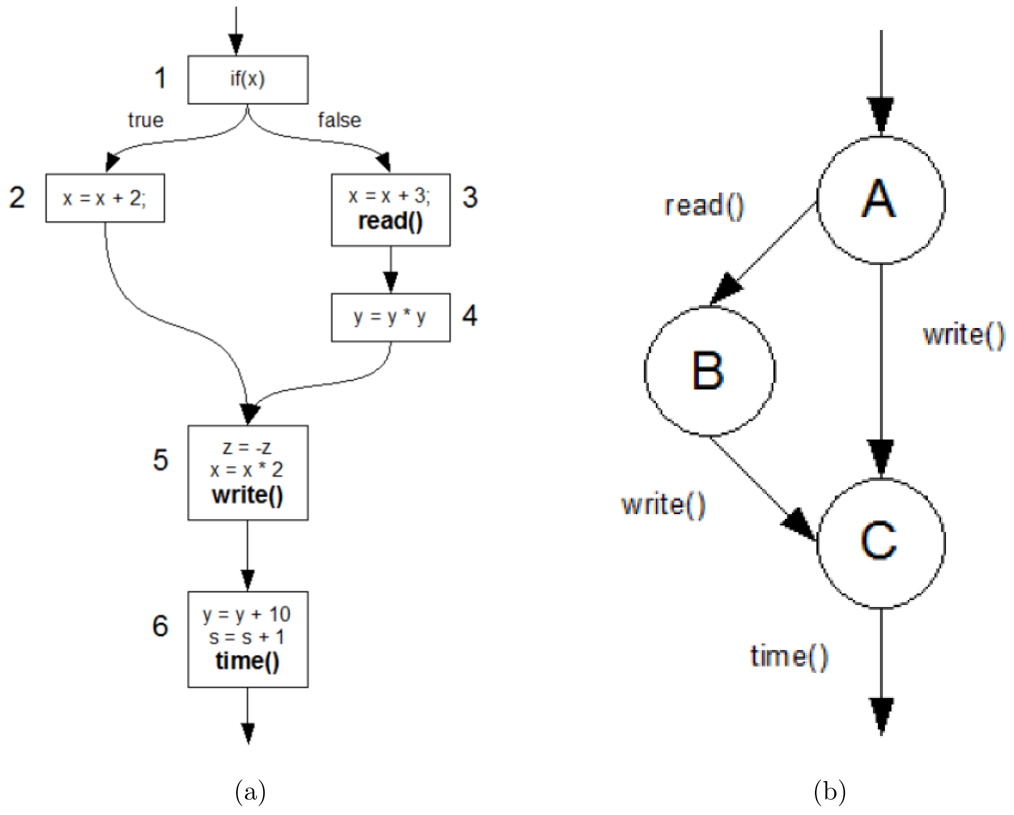


Figure 7.2: Unit Block Graph (a) and System Block Graph (b) for Example 1

The first time, the algorithm returns the set of all the UBs of the first SB (which we call SB A: see Fig. 7.2(b)). SB A contains all the basic blocks met by exploring the control-flow graph until a final UB is found. The UBs of this set are 1, 2, 5, 3, where 5 and 3 are final UBs. The algorithm keeps tracks of the nodes in the graph that are the roots of the next SB, i.e. the UBs following the final UB of the last explored SB (A). These are UBs 4 and 6. Moreover, an arc is added from each final UB of the last SB to each new UB. Then, the algorithm is applied recursively to each new UB.

Then, the algorithm locates the next basic block (4), and the new SB (B) consists of the UBs 4 and 5, where 5 is the final UB. The following block (6) is the only UB of C, the next SB, and it is then explored recursively. At the end, the algorithm returns the following SBs and UBs:

- SB A = {1, 2, 5, 3}, initial UB = {1}, final UBs = {5, 3};
- SB B = {4, 5}, initial UB = {4}, final UB = {5};
- SB C = {6}, initial UB = {6}, final UB = {6}.

Example 2. Consider the following snippet of code:

```
while(x<10){
   y = y + x;
   write();
   x = x+1;
}
z = -z;
read();
time();
```

This is the list of the SBs and of the corresponding UBs:

- SB A = {1, 2, 4}, initial UB = {1}, final UB = {2, 4}.
- SB B = {3, 1, 2, 4}, initial UB = {3}, final UBs = {4, 2}.
- SB C = {5}, initial UB = {5}, final UB = {5}.

Figure 7.3(a) and 7.3(b) show, respectively, the corresponding UBG and SBG.

Example 3. Consider the following snippet of code:

```
while(x<20){
   y = y + x;
   k = k - 2;
   read();
   if(v){
      a = a - x;
      s = a;
```

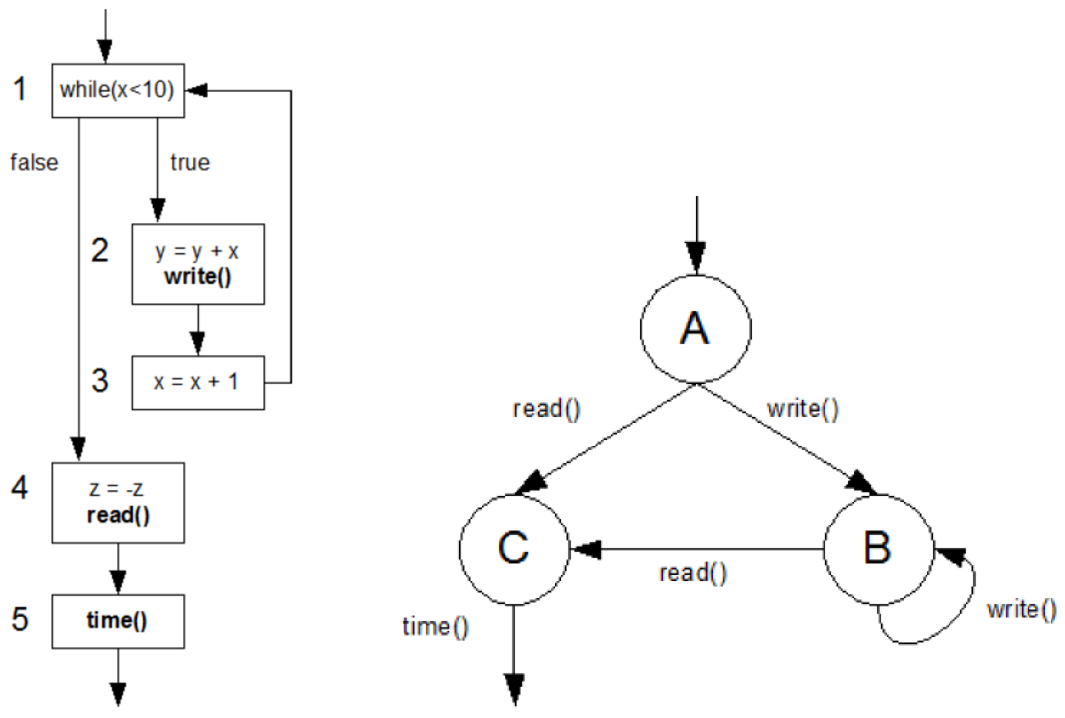


Figure 7.3: Unit Block Graph (a) and System Block Graph (b) for Example 2

```
        write();
    }
    else{
        s = 2;
    }
    x = x + 1;
}
z = -z;
time();
read();
```

This is the list of the system blocks and of the corresponding unit blocks:

- SB A = {1, 2, 7}, initial UB = {1}, final UBs = {2, 7};
- SB B = {3, 4, 5, 6, 1, 2, 7}, initial UB = {3}, final UBs = {4, 2, 7};
- SB C = {6, 1, 2, 7}, initial UB = {6}, final UBs = {2, 7};
- SB D = {8}, initial UB = {8}, final UB = {8}.

Figure 7.4(a) and 7.4(b) show, respectively, the corresponding UBG and SBG.

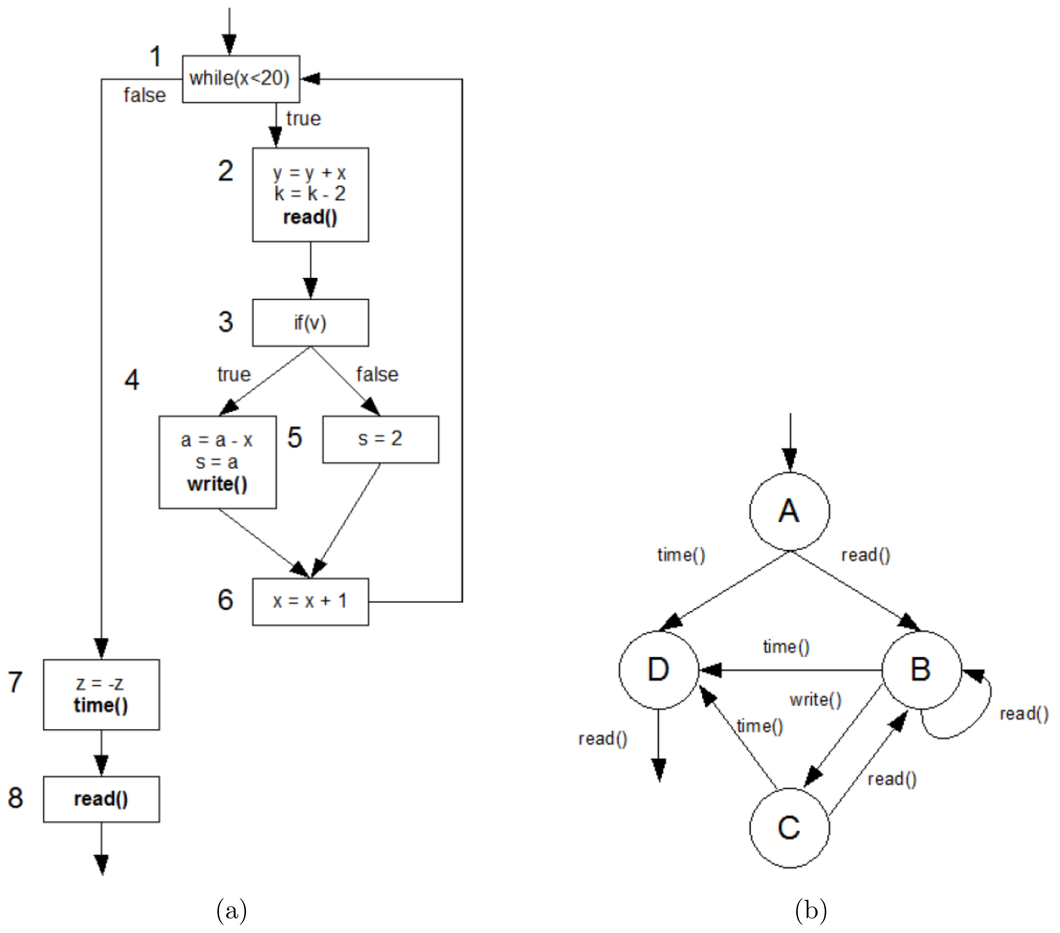


Figure 7.4: Unit Block Graph (a) and System Block Graph (b) for Example 3

7.3 Architecture of the Obfuscation Mechanism

This section describes an architecture for the proposed strategy that exploits virtualization to achieve full transparency for the program to be obfuscated. The overall solution includes two steps:

1. the control-flow logic of the program to be obfuscated is partitioned between two VMs, i.e. the Mon-VM, which runs the SBs of P, and the I-VM, which implements the code encryption/decryption and transfers the control among SBs according to $SBG(P)$; each arc of the $SBG(P)$ is coupled with the virtual address of the corresponding system call;

2. the I-VM encrypts and decrypts SBs at run-time in the memory of the Mon-VM so that only the SB that is currently being executed is in clear.

To increase the complexity for an attacker to rebuild the original code by accessing any of the executed SBs in clear, the Mon-VM only stores the SBs without any information about their execution order. Transfer of control among these blocks is implemented by the I-VM by directly updating the program counter of the Mon-VM. The new value of this register is computed through the SBG and a *jump table* that maps the SBs into their virtual address.

7.3.1 Control-Flow Partitioning

The algorithm described in the previous section supports a partitioning of the control-flow information of a program into two sets: (i) a set of SBs; (ii) the SBG. To produce them, the techniques previously discussed are applied at compile-time to generate: (i) a binary file containing the code of all the SBs in some random order; (ii) one or more files with information about: (a) the SBG, which describes control transfer among the SBs; (b) the SB localizator, which records the initial and final address of each SB in the binary file. At run-time, the SB localizator is used to determine the boundary of the SBs that are encrypted and decrypted. Moreover, it is used as an index to the jump table that records the association among SBs and virtual addresses to retrieve the initial address of the next SB. This address is used to update the program counter of the Mon-VM: in fact, the current value of the program counter is invalid because the SBs have been randomly ordered in the binary file. Any information but the binary file is stored in the I-VM that it protects by the Mon-VM. Only the *Obfuscator* in the I-VM can access the SBG and the jump table. As an example, Table 7.1 describes, at a high-level, the strategy applied by the Obfuscator in the I-VM to protect the program on the Mon-VM described by the SBG in Fig. 7.5.

Any SB consists of a set of UBs. Each UB is coupled with:

- a unique identifier;
- a virtual memory address that is the address of the first instruction of the UB;
- its dimension.

Moreover, also the starting address of the text segment is recorded.

Each node of the SBG is associated with the following data:

- an identifier and the initial virtual address of the SB;
- the set of the UBs that compose the SB;
- the set of the exit points, i.e. the arcs to other nodes. Each exit point includes: (a) an identifier of the next SB; (b) the return address of the system call identifying this arc.

Each transition between two SBs is generated by a system call. At run-time, to distinguish the system call associated with the current arc, we cannot use the system call token, i.e., the name of the system call, because the program may issue the same system call at several distinct program points. For this reason, the Obfuscator uses the virtual address coupled with the system call inside the text segment, which is unique for each system call. In the current prototype, each call is identified by the return address of the invocation, which is the address that immediately follows the call, rather than the system call virtual address. The technical reason is that, at run-time, when a system call is issued and trapped the current program counter does not store the system call virtual address but, instead, the virtual address of the kernel instruction that is executing the system call handler. By retrieving the return address of the system call, which can be found on the stack, we know the virtual address that immediately follows the system call site and, therefore, that of the system call.

```
switch(currentSystemBlock)
{
   case '1' : switch(currentSystemCall)
              { case '1': decrypt(SB2};
                           setPC(jumpTable[firstInstr(SB2)]);
                           encrypt(SB1); break;
                case '2': decrypt(SB3};
                           setPC(firstInstruction(SB3));
                           encrypt(SB1); break;
              } break;
   case '2' : switch(currentSystemCall)
              { case '5': decrypt(SB5};
                           setPC(jumpTable[firstInstr(SB5)]);
                           encrypt(SB2); break;
              } break;
   case '3' : switch(currentSystemCall)
              { case '3': decrypt(SB4};
                           setPC(jumpTable[firstInstr(SB4)]);
                           encrypt(SB3); break;
              } break;
   case '4' : switch(currentSystemCall)
              { case '4': decrypt(SB6};
                           setPC(jumpTable[firstInstr(SB6)]);
                           encrypt(SB4); break;
              } break;
   case '5' : switch(currentSystemCall)
              { case '5': setPC(jumpTable[firstInstr(SB5)]);
                          break;
                case '6': decrypt(SB8};
                          setPC(jumpTable[firstInstr(SB8)]);
                          encrypt(SB5); break;
              } break;
   case '6' : switch(currentSystemCall)
              { case '6': decrypt(SB7};
                          setPC(jumpTable[firstInstr(SB7)]);
                          encrypt(SB6); break;
              } break;
}
```

Table 7.1: Obfuscation Strategy Implemented by Introspection Virtual Machine

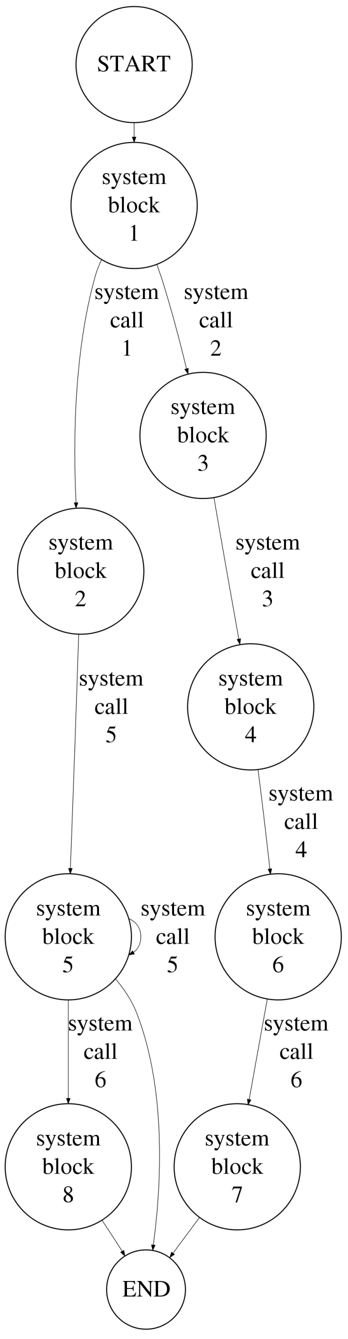


Figure 7.5: System Block Graph

As soon as the Obfuscator knows the return address of a system call, it can:

1. identify transitions between SBs to locate the next SB and retrieve its first address through the jump table;

2. encrypt the current SB and decrypt the next one;
3. redirect the process control-flow to the first instruction of the next SB.

To locate the return address, the Obfuscator in the I-VM:

1. accesses the Mon-VM VCPU context to read the `kernel_sp` register, which points to the top of the kernel stack;
2. maps the Mon-VM kernel stack;
3. reads the `ESP` register, which points to the base of the Mon-VM user stack;
4. maps the Mon-VM user stack;
5. locates the return address of the system call in the user stack of the protected application.

To implement the last step, the Obfuscator has to read the `EAX` register (which contains the system call identifier) not only to identify the system call but also to locate the return address in the stack, since the correct offset from the stack pointer varies among distinct calls. As previously said, the return address is used to identify the transition among SBs. However, in some cases the Obfuscator cannot deduce the next SB from the current system call's return address because the same system call may be coupled with several arcs, e.g. anytime a system call is issued inside a function. In this cases, the Obfuscator scans the stack to detect the SB that has generated the call. Further alternative solutions are: (i) function inlining, i.e. the compiler generates code so that any system call has a distinct address for each case (obviously, this solution cannot be applied in case of recursive procedures); (ii) the compiler generates code to locate the first return address in the stack that is different in all the cases.

7.3.2 Encryption

The binary code is encrypted with keys written in the configuration files stored in the I-VM. Only the first SB is in clear, so that whenever the program is loaded in memory it can be executed until the first system call is executed. When this system call is reached, the Obfuscator in the I-VM can deduce the next SB that will be executed, because it knows the SBG and the current system call, i.e. it knows the current edge in the graph. Hence, the Obfuscator encrypts anything that was in clear before the current system call, decrypts the memory region storing the next SB and updates the program counter to point to the first instruction of this SB. An attacker cannot recover the original program even if she has seen all the previous SBs in clear because she may know some SBs but cannot recover the original program control-flow and the execution order among these SBs that is codified by the SBG in the I-VM.

7.3.3 Run-Time Components

The run-time support of the proposed strategy includes the following components (see Fig. 7.6): the *interceptor module* in the Mon-VM, the *Obfuscator* in the I-VM, which also stores a set of *configuration files*. In the current implementation, the interceptor module is implemented by *HiMod* (see Sect. 5.3.1), a kernel module in the Mon-VM that intercepts the system calls and informs the I-VM that a system call has been executed. It can also be replaced by a hardware virtualization support for trapping interrupt instruction [230].

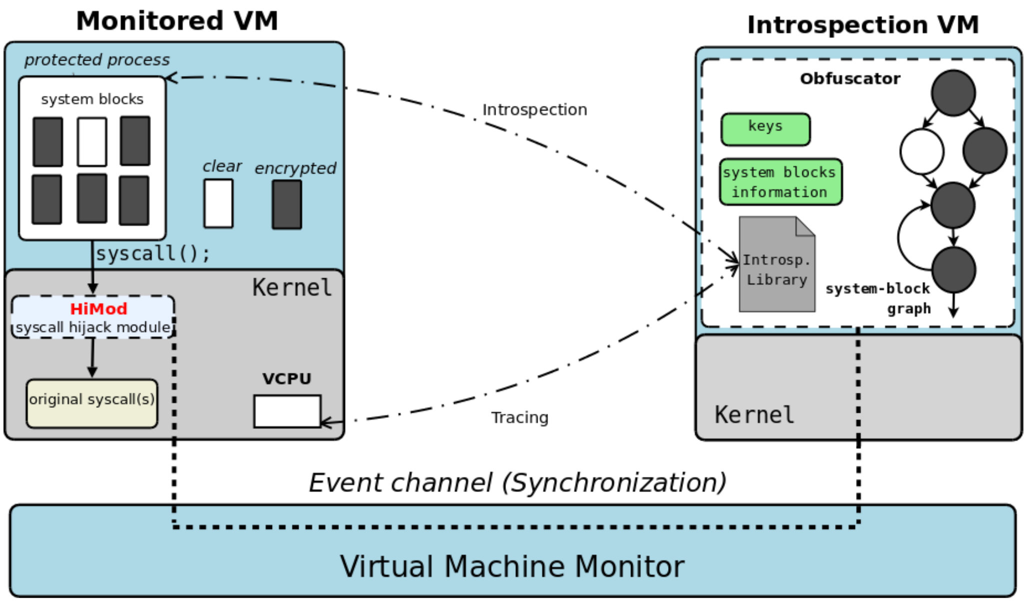


Figure 7.6: Obfuscation Run-Time Architecture

The Obfuscator runs in the I-VM and can access:

- any memory location of the Mon-VM with read and write permissions;
- the information about the flow of the SBG;
- the encryption keys;
- the jump table.

The Obfuscator listens to any events sent by the HiMod and it receives an event anytime the application program invokes a system call. Then, the Obfuscator:

1. freezes the execution of the Mon-VM;
2. deduces the next SB to be executed through the current system call, identified by the current return address and the SBG;

3. updates the Mon-VM program counter to point to the next SB;
4. encrypts the previous SB and decrypts the next one;
5. resumes the execution of the Mon-VM.

The frameworks also includes the configuration files in the I-VM `graph.xml`, `codesegment.xml`, `keys.k`, which store all the information about the SBG, the virtual address of the UBs and the encryption keys. `graph.xml` stores the representation of the SBs and its core element is the SB (`<system>`), which is associated with:

- `<id>`: SB identifier. An integer value greater than or equal to zero, used as identifier of the initial SB;
- `<start_address>`: the virtual address of the first instruction inside the SB;
- `<exit_point>`: it represents an arc to the next SB. An exit point contains:
  - `<return_address>`: the value of the return address identifying the arc;
  - `<next_system_id>`: the identifier of the next SB.
- `<unit_block_id>`: identifier for a UB belonging to the SB.

The following is an example of a `graph.xml` file:

```
<application_graph>
  <system>
    <id> 0 </id>
    <start_address> 80483a4 </start_address>
    <exit_point>
      <return_address> 80483df </return_address>
      <next_system_id> 1 </next_system_id>
    </exit_point>
    <unit_block_id> 0 </unit_block_id>
  </system>


  <system>
    <id> 1 </id>
    <start_address> 80483f7 </start_address>
    <exit_point>
      <return_address> 804849c </return_address>
      <next_system_id> 2 </next_system_id>
    </exit_point>
    <exit_point>
      <return_address> 804849c </return_address>
      <next_system_id> 3 </next_system_id>
    </exit_point>
    <unit_block_id> 1 </unit_block_id>
    <unit_block_id> 2 </unit_block_id>
    <unit_block_id> 3 </unit_block_id>
  </system>

  <system>
```

```
        <id> 2 </id>
        <start_address> 80484a1 </start_address>
        <exit_point>
            <return_address> 80485e6 </return_address>
            <next_system_id> 3 </next_system_id>
        </exit_point>
        <unit_block_id> 4 </unit_block_id>
    </system>

    <system>
        <id> 3 </id>
        <start_address> 80485eb </start_address>
        <exit_point>
            <return_address> 8048870 </return_address>
            <next_system_id> 4 </next_system_id>
        </exit_point>
        <unit_block_id> 5 </unit_block_id>
        <unit_block_id> 6 </unit_block_id>
    </system>


    <system>
        <id> 4 </id>
        <start_address> 8048875 </start_address>
        <exit_point>
            <return_address> 8048d7a </return_address>
            <next_system_id> 5 </next_system_id>
        </exit_point>
        <unit_block_id> 7 </unit_block_id>
        <unit_block_id> 8 </unit_block_id>
        <unit_block_id> 9 </unit_block_id>
        <unit_block_id> 10 </unit_block_id>
    </system>

    <system>
        <id> 5 </id>
        <start_address> 8048d7c </start_address>
        <exit_point>
            <return_address> 8048d89 </return_address>
            <next_system_id> 6 </next_system_id>
        </exit_point>
        <unit_block_id> 11 </unit_block_id>
    </system>

    <system>
        <id> 6 </id>
        <start_address> 8048d89 </start_address>
        <unit_block_id> 12 </unit_block_id>
    </system>

</application_graph>
```

The second file, `codesegment.xml` records the virtual address of each UB, that of the code segment and the encryption algorithm used, and it contains the following fields:

- `<CSstart_address>`: the initial code segment address;
- `<CSend_address>`: the final code segment address;
- `<encryption_method>`: the name of the encryption algorithm;

- `unit_block`: it represents a UB, and it contains:
  - `<id>`: identifier for the UB;
  - `<initial_address>`: initial virtual address of the UB;
  - `<size>`: length of the UB.

The following is an example of `codesegment.xml`:

```
<code_segment>

   <CSstart_address> 8048320 </CSstart_address>

   <CSend_address> 8048d97 </CSend_address>

   <encryption_method> DES </encryption_method>

   <unit_block>
      <id> 0 </id>
      <initial_address> 80483a4 </initial_address>
      <size> 59 </size>
   </unit_block>

   <unit_block>
      <id> 1 </id>
      <initial_address> 80483f7 </initial_address>
      <size> 165 </size>
   </unit_block>

   <unit_block>
      <id> 2 </id>
      <initial_address> 80484a1 </initial_address>
      <size> 325 </size>
   </unit_block>

   <unit_block>
      <id> 3 </id>
      <initial_address> 80485eb </initial_address>
      <size> 645 </size>
   </unit_block>

   <unit_block>
      <id> 4 </id>
      <initial_address> 8048875 </initial_address>
      <size> 1285 </size>
   </unit_block>

   <unit_block>
      <id> 5 </id>
      <initial_address> 8048d7c </initial_address>
      <size> 13 </size>
   </unit_block>

   <unit_block>
      <id> 6 </id>
      <initial_address> 8048d89 </initial_address>
      <size> 14 </size>
   </unit_block>
</code_segment>
```

Finally, `keys.k` records the keys to encrypt each UB, ordered by the UB they belong to. We recall that the I-VM may receive this file at run-time as well, after the successful execution of some acceptance test on the underlying system.

7.4 Performance Results

Using a CPU Intel code 2 Duo T7500 2.2. GHz with 3GB DDR2 RAM, we tested the latency of the following mechanisms: (i) system call interception; (ii) location of the return address; (iii) SB transition; (iv) SB encryption; (v) SB decryption.

The interception of a system call requires several steps: (i) the trap of the current system call invocation by the HiMod; (ii) the communication of the HiMod to alert the Obfuscator. Hence, the resulting time is the sum of the system call interception latency, the channel event latency to inform the Obfuscator and the time spent by the HiMod waiting for the notification by the Obfuscator. The tested average value for this time is:

$$C_{int} = 52\mu sec$$

As previously said, to locate the return address the I-VM has to (i) retrieve the value of a register; (ii) unwind two stacks. If C_{get} denotes the time of this computation, then:

$$C_{gret} = (a \cdot x) + d \quad [+a \cdot y]$$

where x is the number of kernel-stack frames to traverse, a is the cost to traverse a stack frame, d is the cost to read the `EBP` register, the pointer to the current frame, retrieve the value of the saved register at the bottom of the kernel-stack and analyze the top of the user-stack, y is the number of user-stack frames to be scanned if the return address to deduce the next block is not in the top frame of the user-stack. If the optional part is not required (i.e., $y = 0$), the average time for this operation is:

$$C_{gret} = 81\mu sec$$

To implement a SB transition, the I-VM considers the node representing the current SB and searches the arc of the SBG identified by the return address located in the previous step. Then, it updates some internal states and modifies the program counter to point to the virtual address of the next SB. The average time is:

$$C_{tstats} = 4\mu sec$$

To encrypt a SB, the I-VM first retrieves the keys associated with the block and then encrypts it. This requires a time that can be computed as follows:

$$C_{crypt} = \sum_{i=1}^{n}(c_i + u_i) + g$$

where n is the number of UBs in the SB; c_i and u_i are, respectively, the cost of encrypting the i-th UB and of copying the i-th UB into the corresponding memory location; g is the fixed cost to manage the remaining computations. In more detail, c_i can be computed as:

$$c_i = s + (l_i \cdot e_i)$$

where s is the fixed cost for managing each UB; l_i is the length of the i-th UB; e_i is the cost of encrypting a unit of memory. Instead, u_i is computed as:

$$u_i = l_i \cdot k_{um}$$

where l_i is the length of the i-th UB; k_{um} is the cost of writing a unit of memory.

The tests produced the following average times:

$$\begin{aligned} g &= 1\mu sec \\ s &= 41\mu sec \\ k_{um} &= 0.01\mu sec \\ e_i &= 0.05\mu sec \quad \text{(for byte, using DES)} \\ e_i &= 0.036\mu sec \quad \text{(for byte, using Blowfish)} \\ e_i &= 0.008\mu sec \quad \text{(for byes, using One Time Pad)} \end{aligned}$$

If a SB includes exactly one UB with a size of 1 KB, then:

$$\begin{aligned} C_{crypt} &= 103\mu sec \quad \text{(using DES)} \\ C_{crypt} &= 89\mu sec \quad \text{(using Blowfish)} \\ C_{crypt} &= 60\mu sec \quad \text{(using OTP)} \end{aligned}$$

To decrypt a UB, the I-VM retrieves the key associated with the corresponding SB. The formulas to compute the associated costs are the same ones of the previous case and the previous results still hold.

The overhead associated with a system call is:

$$C_{int} + C_{gret} + C_{tstat} + C_{crypt} + C_{decrypt}$$

The value of the first three parameters is fixed and is given by:

$$C_{int} + C_{gret} + C_{tstat} = 52\mu sec + 81\mu sec + 4\mu sec = 137\mu sec$$

If we consider the previous example of a SB including exactly one 1k UB, we have:

$$\begin{aligned} C_{crypt} + C_{decrypt} &= 103\mu sec + 103\mu sec = 206\mu sec \quad \text{(using DES)} \\ C_{crypt} + C_{decrypt} &= 89\mu sec + 89\mu sec = 178\mu sec \quad \text{(using Blowfish)} \\ C_{crypt} + C_{decrypt} &= 60\mu sec + 60\mu sec = 120\mu sec \quad \text{(using OTP)} \end{aligned}$$

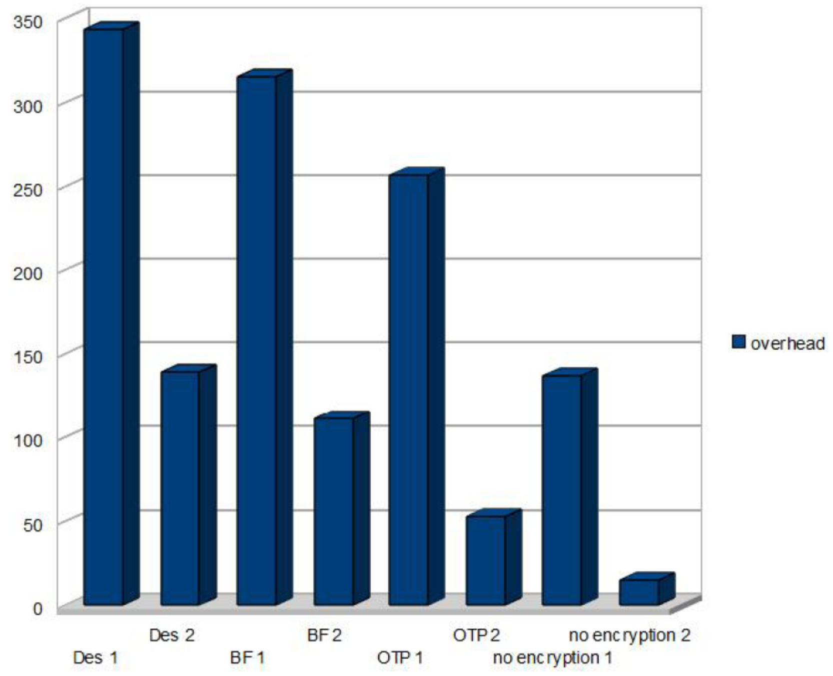


Figure 7.7: Protection Overhead: 1) Current Library; 2) Extended Library

If we sum the previous results, we have:

$$\begin{aligned}
C_{tot} &= 137\mu sec + 206\mu sec = 343\mu sec \quad \text{(using DES)}\\
C_{tot} &= 137\mu sec + 178\mu sec = 315\mu sec \quad \text{(using Blowfish)}\\
C_{tot} &= 137\mu sec + 120\mu sec = 257\mu sec \quad \text{(using OTP)}
\end{aligned}$$

Obviously, to compute the total latency, the time to perform a system call should be added to these results. Since in the current prototype the Introspection Library cannot map two pages simultaneously, whenever the I-VM needs to access a page, the Introspection Library unmaps the previous mapped page and maps another one. Since the cost of a map operation is 37μsec and of an unmap is 4μsec, the extension of the Introspection Library to map several pages concurrently results in the following performance benefits:

- system call interception: this operation maps and unmaps the page corresponding to the synchronization channel. Hence, by avoiding these two operations, the new cost is:

$$C_{int} = 52 - (C_{map} + C_{unmap}) = 11\mu sec$$

- locating the return address: this operation maps two pages storing the kernel and user stacks and unmaps the kernel stack. Thus, by avoiding these operations, the cost is:

$$C_{gret} = 81 - (2 \cdot C_{map} + C_{unmap}) = 4\mu sec$$

- SB transition: the cost is negligible, since the user stack is not unmapped;
- the cost of map and unmap depends upon that to manage each UB. Also this cost is negligible if we neglect the costs of map and unmap.

As an example, if the Introspection Library can map several pages concurrently, the costs for a SB composed of exactly one 1k UB are:

$$\begin{aligned}
C_{crypt} = C_{decrypt} &= 62\mu sec \quad \text{(using DES)}\\
C_{crypt} = C_{decrypt} &= 48\mu sec \quad \text{(using Blowfish)}\\
C_{crypt} = C_{decrypt} &= 19\mu sec \quad \text{(using OTP)}
\end{aligned}$$

and the fixed overhead associated with each system call invocation is:

$$C_{int} + C_{gret} + C_{tstat} = 11\mu sec + 4\mu sec + 0\mu sec = 15\mu sec$$

and we have that:

$$\begin{aligned}
C_{tot} &= 15\mu sec + 2 \cdot 62\mu sec = 139\mu sec \quad \text{(using DES)}\\
C_{tot} &= 15\mu sec + 2 \cdot 48\mu sec = 111\mu sec \quad \text{(using Blowfish)}\\
C_{tot} &= 15\mu sec + 2 \cdot 19\mu sec = 53\mu sec \quad \text{(using OTP)}
\end{aligned}$$

Fig. 7.7 shows the overhead in the all the previous scenarios.

Chapter 8

Trusted Overlays of Virtual Communities

This chapter introduces *Virtual Interacting Network CommunIty* (Vinci), a software architecture that exploits virtualization to share in a secure way an information and communication technology (ICT) infrastructure among a set of users with distinct security levels and reliability requirements. To this purpose, Vinci decomposes users into *communities*, each consisting of a set of users, their applications, a set of services and of shared resources. Users with distinct privileges and applications with distinct trust levels belong to distinct communities. Each community is supported by a virtual network, i.e. a structured and highly parallel overlay that interconnects VMs built by instantiating one of a predefined set of *VM templates*. Some VMs run user applications, some protect shared resources and some others control traffic among communities to filter out malware or distributed attacks. Further VMs manage the infrastructure resources and configure the VMs at start-up. The adoption of alternative VM templates enables Vinci to minimize the complexity of each VM and increases the robustness of both the VMs and of the overall infrastructure. Moreover, the security policy that a VM applies depends upon the community a user belongs to. As an example, discretionary access control policies may protect files shared within a community, whereas mandatory policies [171] may rule access to files shared among communities.

8.1 Introduction

While the most well-known benefit of virtualization is the cost saving achieved by server consolidation [238], the previous chapters have discussed a further noticeable advantage, namely an increase of system robustness. This is due to the low cost of including into a virtual architecture some components that check and control the other ones in a transparent way. As an example, an overlay can include VMs that run the applications and distinct VMs that monitor the VMs running the applications in a completely unobtrusive way, such as the Assurance VMs in VIMS.

A further, distinct, advantage of virtualization is that an overlay including a large number of VMs can increase the robustness of each VM, and of the overall system, by minimizing the software each VM runs. This implies that the number of virtual nodes, i.e. of VMs, may be rather large with respect to the number of physical nodes. In this way, we can optimize error confinement at the expense of the virtualization overhead. However, the latter overhead may be strongly reduced provided that proper hardware support is available, as it is often the case in the last generation processors. These considerations have led to the definition of *Virtual Interacting Network CommunIty* (Vinci), a software architecture that aims to exploit at best virtualization technologies to share in a secure way an ICT infrastructure. To this purpose, Vinci adopts a two-tier approach that introduces several overlays and each overlay is highly parallel because it includes a large number of VMs. To increase the robustness of an overlay, Vinci minimizes the functionalities of each VM by defining several VM templates. As an example, Vinci instantiates Application VMs to run user applications, according to the applications trust level and to the user privileges, i.e. user security levels, so that each Application VM only runs the smallest number of software packages and libraries to support the considered applications. Other VM templates are introduced to control resources shared among Application VMs of the same overlay or of distinct ones, or information flowing among overlays.

The number of overlays that are mapped onto the infrastructure depends upon user *communities*, because a distinct overlay, or *virtual community network* (VCN), is introduced for each community. A community consists of a set of users that execute applications and of services that these applications exploit. The users and applications in a community can be handled in a uniform way because they have homogeneous security and reliability requirements. Communities can also cooperate and exchange information. Proper consistency and security checks are applied within a community, while more severe checks are enforced to cross the community border. When defining a community, an administrator pairs it with a *global level*, which defines the set of users that can join the community, the applications they can run and the resources they can access. In this way, the global level is the same for all the VMs in a community and they can be homogeneously managed because they have similar requirements. Hence, the notion of community simplifies the overall management of the VMs, because VMs of the same overlay require the same global level and the data they exchange can be protected through the same mechanisms.

Example. An example of an infrastructure where Vinci can be applied is the one of a hospital that is shared, at least, among the doctor community, the nurse community and the administrative community. Since each of these communities manages its private information but also shares some data with the other ones, it should be associated with its reliability requirements, its security policy and with controls on information that may be shared with the other ones. As an example, members of the doctor community can update information about prescriptions whereas those in the nurse community can read but not update the same information. Both the nurse community and the doctor community share some data with the administrative community, which has to bill the patient insurances. In the most general case, each user belongs to several communities according to the applications she needs to run and the data she wants to access. Consider a doctor that is the head of the hospital: as a doctor she belongs to the doctor community but, because of her administrative duties, she belongs to the administrative community as well. Furthermore, the community the doctor joins to access critical health information differs from that she joins when surfing the Internet.

8.2 Virtual Interacting Network Community Architecture

In the general case, the infrastructure architecture is a private network that spans several locations, it includes a rather large number of physical nodes and it is centrally managed by a set of administrators. We also assume that most of the nodes of the infrastructure are personal computers that can only be accessed by one person at a time and that the infrastructure includes a set of servers to store data shared among communities and execute server applications. Vinci requires that each node runs a VMM, which guarantees both the confinement among the VMs and a fair access to the node's resources.

As said before, one of the main advantages of virtualization is the ability of choosing the appropriate combination of OS and applications for each VM. To exploit at best this feature, Vinci defines a set of highly specialized and simple *VM templates* that are dynamically instantiated and connected into overlays, i.e. *virtual community networks* (VCNs). A Vinci VCN includes both VMs that run applications and VMs that support and monitor the previous ones. While a VCN strongly resembles a virtual private network (VPN), an important difference lies in the granularity of the computation because when defining a VCN we are interested in minimizing the complexity of the services that each VM implements.

In Vinci, each VCN is built by connecting VMs that are instances of the following templates:

1. *Application VM* (APP-VM): it runs a set of applications on behalf of a single user;

2. *Storage VM* (STO-VM): it exports a shared storage. It is further specialized in:

   - *Community VM* (COM-VM): it manages the private resources of a community by enforcing mandatory and/or discretionary access control (MAC/-DAC) policies;

- *File System VM* (FS-VM): it belongs to several VCNs to protect files shared among the corresponding communities. It can implement MAC and Multi-Level Security policies and a tainting mechanism to prevent illegal information flows across communities.

3. *Communication and Control VM* (CC-VM): it implements and monitors information flows among communities, i.e. flows among CC-VMs of distinct communities, or private flows among VMs of the same community;

4. *Assurance VM* (A-VM): it checks that APP-VMs only run authorized software and attests the software of a VM.

Moreover, Vinci introduces *Infrastructure VMs* (INF-VMs) that do not belong to any VCN and extend the VMMs with new functionalities to manage the overall infrastructure. As shown in Fig. 8.1, VMs that are instances of the same template have homogeneous requirements and system configurations: thus, they are easy-to-deploy virtual appliances created on demand from a generic baseline image. Moreover, when a VM is instantiated, its run-time environment is highly customized according to the user and the community of interest through a set of parameters that include, among others, the amount of memory, the running kernel modules, and the OS and applications versions.

In the following, we describe the current implementation of Vinci that exploits Xen [26] to create the VMs and connect them into VCNs. NFSv3 and Security-Enhanced Linux (SELinux) [159, 158] have been modified to apply security policies based upon the security levels of users or the global levels of communities. Finally, interconnections among VMs are handled through iptables [167] and OpenVPN [173].

8.2.1 File Sharing

The application model of interest consists of a set of application processes executed by several users, where each process P can access some files in one or more shared file systems, denoted as $FS_1(P), \ldots, FS_n(P)$. We assume that $T(P)$, the trust level coupled with P, is known and that it also holds for all the processes spawned by P.

To execute the application processes and export the file systems, the application model introduces a cluster of VMs. Any VM that exports a shared storage belongs to the class of Storage VMs (STO-VMs). STO-VMs implement a highly secure storage through a *file sharing server* (FSS) module configured according to the security policy defined for the corresponding file system. Each APP-VM runs a *file sharing client* (FSC) module that acts as a proxy for the application processes and interacts with the proper FSS module, so that application processes can access any files unaware of the type and the location of the file system.

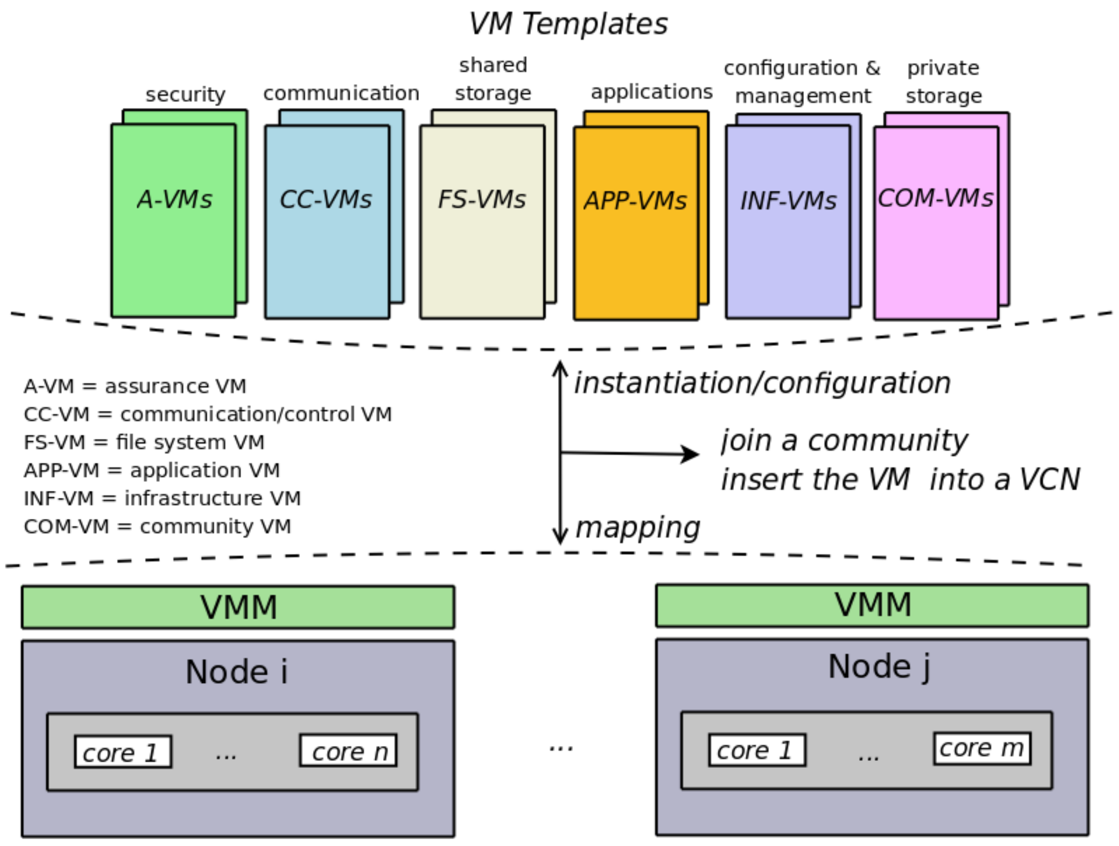


Figure 8.1: Virtual Machine Templates

The mapping of applications and file systems onto the physical architecture is implemented in two steps:

1. the first step maps the application processes onto APP-VMs and each shared file system onto a distinct STO-VM;

2. the second step maps all the VMs (APP-VMs and STO-VMs) onto the physical machines.

The goal of the first mapping is to minimize the sharing among the application processes. The association of application processes and APP-VMs is static because an application process cannot migrate from an APP-VM to another one, since processes executed by users with distinct security requirements, i.e. with distinct global levels, should not be mapped onto the same APP-VM. On the other hand, the mapping in the second step is dynamic and an INF-VM can migrate at run-time an APP-VM or an STO-VM to another physical node (see Fig. 8.2). The goal of this step is to balance the computational load among the machines and the communication load among the components the interconnection structure.

To show how an application can access a file, consider an application process P on APP-VM(P), which locally mounts any shared file system $\mathrm{FS}_i(P)$ exported by

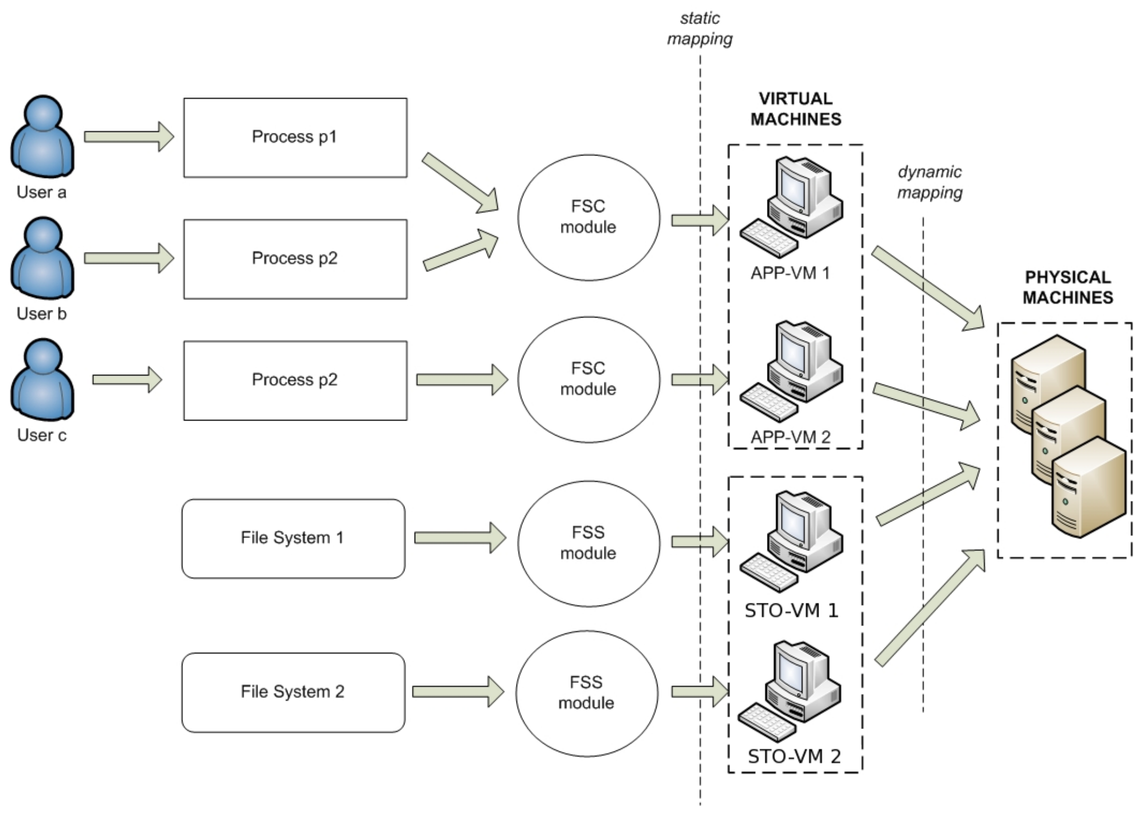


Figure 8.2: Abstract Application Model

STO-VM$_i$(P). While the local file system of APP-VM(P) handles any operation on the private files of P, any request of P to access a file in FS$_i$(P) is trapped and transferred to the FSC module of APP-VM(P) that transmits it to the FSS module on STO-VM$_i$(P). This module checks the request and serves it only if it satisfies the security policy of FS$_i$(P), and then returns the result to FSC. P is unaware that it is accessing a file on a distinct VM.

If we consider file sharing as a specific case of resource sharing, most of the previous concepts applies to the more general context too.

8.2.1.1 Threat Model

The threat model refers to the cloud model of Infrastructure-as-a-Service [240] and it is focused on attacks implemented by application processes against processes of other overlays. The application process may act on behalf of a malicious user or of an insider to implement attacks with the goal of accessing some shared resources, such as files in the considered model. The attacks may be implemented either by a malicious application or by malware code injected into an application by a previous attack. The threat model can also cover attacks against a user application of the same community because an overlay may include VMs to detect these attacks by monitoring the behavior of other VMs and the data they exchange. The number of these VMs and the complexity of their checks depend upon the trust level of the users and of the applications they run. A further assumption underlying the threat model is that the VMMs and the A-VMs belong to the Trusted Computing Base (TCB). The number of VMs to be trusted can be minimized by applying the attestation strategies discussed in Chap. 6. This threat model describes cases where a private or a community cloud [170] is shared among several communities that trust the cloud provider but do not trust each other. However, since the cloud provider is trusted, malicious users cannot implement physical attacks against the infrastructure to sniff or alter the information flowing among the physical nodes. On the other hand, virtual connections among the VMs of an overlay can be attacked. To prevent these attacks, the IP address of each VM is statically assigned and known so that the consistency of communications among the VMs may be preserved by preventing a VM to spoof an address.

As an example, this threat model describes in a realistic way a trusted Intranet executing a set of untrusted applications, where each application can implement attacks to export some information that it illegally accesses. The Intranet connections are trusted because we can assume that a physical attack against the infrastructure to sniff or alter the the information flowing among the physical nodes is rather complex.

8.2.1.2 Security Policy: Implementation

To implement in a transparent way the security policy that controls the file sharing, each APP-VM executes a FSC module that remotely accesses a FSS module on an STO-VM. In turn, the FSS module delegates the security policy description and enforcement to a *MAC-based Security* (MAC-S) module, which guarantees the fulfillment of critical security requirements, such as the integrity and the confidentiality of the shared storage. The adoption of the MAC-S module enables Vinci to support a large set of MAC or DAC class policies on the shared storage and reduces the impact of successful attacks. As an example, MAC allows Vinci to reduce the privileges associated to the superuser on the APP-VMs and to minimize the impact on the shared storage resulting from flawed or malicious application process on an APP-VM.

Every file request from FSC to FSS includes the IP address of the APP-VM that produces the original request. The MAC-S module uses the address to protect the shared storage through a default-deny approach where an application can only access those files for which the current policy grants an authorization. To associate a protection domain with each APP-VM, the MAC-S module identifies the APP-VM through the source IP address in the request and labels the APP-VM with a security context according to the set of privileges associated with the trust level coupled with the IP address. Therefore, if the security policy is parametric with respect to the trust level of the APP-VM, the user that executes an application process can inherit the protection domain of the APP-VM that runs the process. In this way, all the users of an APP-VM can access the same files with the same privileges, whereas users on APP-VMs with distinct trust levels can access distinct sets of files. Hence, the granularity level of the overall security policy can be tuned by updating the number of APP-VMs. A more detailed model of the implementation of an STO-VM will be described in the following.

8.2.1.3 Security Policy: Assurance

Vinci can integrate PsycoTrace components to detect attacks against the VMs. This requires that each node runs an A-VM that analyzes the memory of VMs onto the same node through PsycoTrace run-time tools to discover attacks against the kernel or the processes of these VMs. Furthermore, both an APP-VM and an STO-VM may run some IDS agents to detect intrusions against the local OS or some application processes. All the agents on the VMs on the same physical node are connected to the A-VM through a dedicated virtual control network that delivers the agent alerts.

PsycoTrace tools are even more powerful when applied to an STO-VM because of the low degrees of freedom of the configuration of this VM that only runs a fixed set of processes, which use a fixed set of storage resources and so on. Because of these constraints, the A-VM can apply to any STO-VM the most severe strategies to describe the process self of its applications.

8.2.2 Application Virtual Machines

Each APP-VM runs applications of a single user and is coupled with the global level inherited from the corresponding community. In general, the resources and services that an APP-VM can access depend upon the user security level and the global level of the community that the user of the VM belongs to. Since a user can join distinct communities through distinct APP-VMs, she can access distinct resources/services according to the global level of each community. In some cases, there may exist some resources that a user can access regardless of the community she currently belongs to. As an example, each user can always access its private files. While the global level of a community constrains the users that can join the community and the applications they can run, an administrator can also dictate which resources a community can access and/or share with other communities. Thus, when a user wishes to join a community, and she has the rights to do so, the community statically defines the set of applications that the APP-VM can run and the resources it can access.

In Vinci, during the login phase, a user chooses the community she wants to join. Then, the INF-VM of the local node configures and starts up a corresponding APP-VM to run those applications enabled by the community policy and it connects the APP-VM to the proper VCN. A user can run several APP-VMs on the same node concurrently, each belonging to the same community or to distinct ones.

In the current prototype, each APP-VM is associated with a minimal partition on one of the disks in the physical node, which stores the OS kernel loaded during the boot-up of the APP-VM. Other files may be stored either locally, in a COM-VM in the same VCN, or in a FS-VM shared with other VCNs. To simplify the implementation of security policies, at boot time an IP address is statically assigned to an APP-VM and both the VM global level and the user security level are coupled with this address. Since the IP address uniquely determines the resources the VM and the user can access, CC-VMs implement proper checks to detect any spoofed traffic in a VCN.

8.2.3 Storage Virtual Machines

STO-VMs are generic VMs that export a shared storage and they are implemented by COM-VMs and FS-VMs. A Vinci VCN always includes at least one COM-VM to manage and control the resources shared within the corresponding community, i.e. among its APP-VMs only. A COM-VM stores the community private files, which include configuration files, system binaries, shared libraries and user home directories. In the current prototype, COM-VMs protect the files that they manage through MAC and DAC security policies where the subject of the policy is the user security level, which is deduced from the IP address of the requesting APP-VM. On the other hand, a FS-VM that belongs to several VCNs supports file sharing across communities. The security policies that this VM enforces extend those of a COM-VM by considering both the security level and the community of a user.

In the following, we describe the extensions to common components of FS-VMs and COM-VMs. Then, we discuss the extensions applied to the FS-VM kernel only to insert a *Tainting module* in-between the NFS server and the Virtual File System.

NFSv3 Overview. The NFS service implements a distributed file-system based upon a client-server architecture, by exporting to the clients one or more directories of the shared file system. According to the general model previously described, each APP-VM executes one and only one NFSv3 client, i.e. the FSC module, and every STO-VM executes both an NFSv3 server, i.e. the FSS module, and an SELinux module, i.e. the MAC-S module.

Currently, NFS servers exploit the information in each RPC request generated by the client to authorize or deny access to the shared files, according to the server OS DAC class policies. Since, according to our threat model, client VMs represent a source of untrusted information, NFS has to be modified because it essentially trusts the client machines, and it enables an attacker to maliciously impersonate a legitimate user on an APP-VM with little effort [172].

SELinux Overview. SELinux implements MAC policies through a combination of type enforcement (TE), role-based access control (RBAC) and Identity-based Access Control (IBAC). The TE model assigns types to every OS object, such as files, processes, and network connections. In this way, the security policy can define the rules governing the interactions among OS objects, by implementing a fine-grained access control that satisfies the least privilege principle [206]. SELinux is based upon the Linux Security Modules (LSM) [220, 255], a patch for the Linux kernel that inserts both security fields into kernel data structures and calls to specific hooks into security-critical kernel operations to manage the security fields and to implement access control.

When the SELinux policy is configured, an administrator can label every kernel component with a security context. Processes, identified by a *domain*, are the subjects of the SELinux policy. At run-time, the security policy can associate the subject with its privileges to grant or deny access to system objects according to the requested operation. The policy description specifies both the programs a process can execute and the legal domain transitions.

The idea of integrating NFS with SELinux stems from the need to centrally control client accesses to the shared files and to assign distinct privileges to each APP-VM, leveraging the SELinux flexibility to describe MAC policies.

Modifications to NFS and SELinux. We have modified the Linux kernel to enable an STO-VM to exploit a simpler resource sharing management and a fine-grained access control mechanism. A compile-time option enables the administrator to configure the kernel of the STO-VM to integrate SELinux and NFS. On the other hand, no modifications are required on the APP-VMs.

Through the modified SELinux labeling and access rules, STO-VM administrators can manage, from a single central point, the indirect accesses of an NFS client to the shared file systems. As an example, an administrator can couple each NFS client with a security context and, if proper privileges are assigned to this context, Vinci can satisfy the least privilege principle without sacrificing transparency.

Moreover, since the A-VM can identify the IP address of each APP-VM in a reliable way, STO-VM considers the APP-VM that generated the file request as the real subject of the current security policy.

NFS Client Subject. SELinux labeling and access rules have been extended to introduce a new subject corresponding to the NFS client and to define all the operations it can invoke. In turns, this requires the extension of the SELinux network object *node* [163], by adding into the corresponding object class the operations executed by the NFS server on behalf of NFS clients, such as read, write and create files or directories. In general, nodes are used to control network traffic, i.e. to grant or deny a process the permissions to exchange data with a specific IP address through the network interfaces, and are associated with an IP address and a net-mask through the `nodecon` SELinux syntax statement.

These extensions allow Vinci to define a distinct protection domain for each NFS client and to dynamically associate the NFS server process with the security context of the NFS client requesting the operation.

NFS Request. To properly describe the extensions to the kernel modules, we consider the flow of a request from a NFS client to the NFS server and show how the data structures have been extended and where the modified functions are invoked.

SELinux stores run-time security information about the kernel objects in some data structures, such as tasks, i-nodes and files. The main structure that stores security information about the running processes is `task_security_struct`.

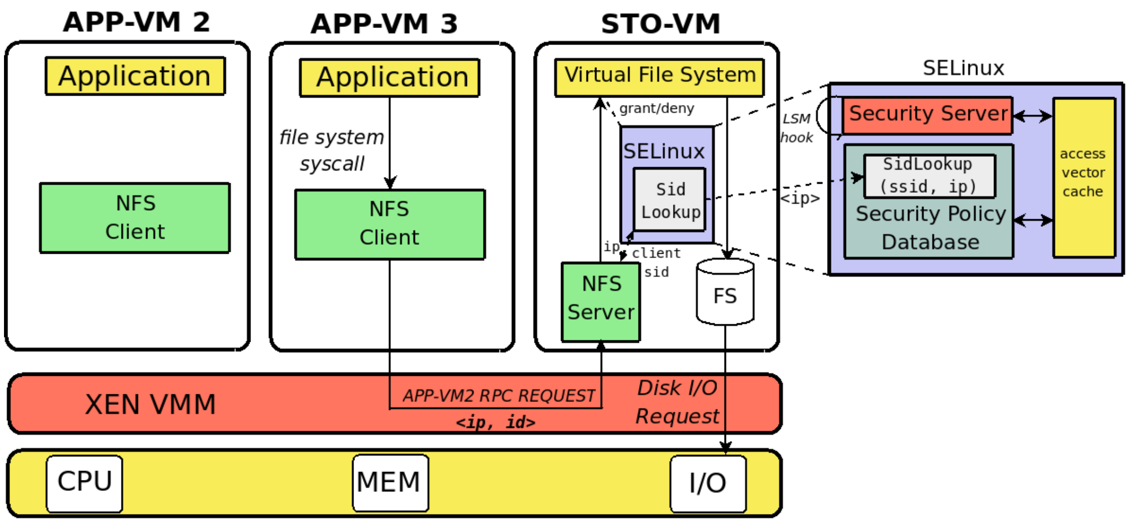


Figure 8.3: File Request

`NfsSid` is a new field of `task_security_struct` that we have added to represent the security identifier bound to the node type. This field is coupled with the IP address and the net-mask of the NFS client that is trying to access the shared file. Every time the NFS server processes a request, the RPC service invokes a new SELinux function (called *SidLookup* in Fig. 8.3) that maps the NFS server SID (SSID) and the NFS client IP address into a SID according to the SELinux Security Policy Database, i.e. the database that stores the current SELinux policy. If the current security policy maps the requesting IP address with a node type, *SidLookup* returns the corresponding SID and the related security context, otherwise it returns a default unprivileged SID. Before the NFS server invokes the system call on the file system, the NFS client SID is copied into the `NfsSid` field of `task_security_struct` of the NFS daemon process servicing the request. Later, when the NFS server invokes the system call to access the shared file system on behalf of NFS clients, the kernel triggers an LSM hook to delegate security controls to SELinux.

LSM Hook Modifications. The appropriate LSM hooks have been modified to enforce access controls on the operations a subject can invoke. The controls are applied when the NFS server:

- uses a capability;
- updates an i-node;
- updates a file;
- creates or removes a file, a directory, a link to a file or to a directory;
- renames a file or a directory;
- operates on the file system (super-block).

We have modified all these hooks so that two cases are considered. If the NFS server task is coupled with the SID relative to the IP address of the requesting NFS client, then the SELinux Security Server applies the current security policy by considering the `NfsSid` as the policy subject. If, instead, an unprivileged default SID relative to the node type is assigned to the `task_security_struct`, then the subject of the SELinux controls is the NFS server daemon process. After the kernel has handled the system call, the `NfsSid` field is reset to a default value corresponding to an unprivileged domain. In both cases, the Security Server authorizes or denies file access according to the current security policy. In this way, NFS clients own some privileges on the remote file system exported by the STO-VM just for the time interval to serve client requests on a shared file.

8.2.3.1 Tainting Module

FS-VMs exploit the capabilities of a *Tainting module* to prevent the flow of information among predefined communities. This module can:

(i) confine information flows among communities;

(ii) increase the robustness with respect to contamination attacks;

(iii) log the actual flow of information among communities.

To this end, the Tainting module associates each user of a community with a bit mask, i.e. the community mask, which, by default, has exactly one bit set to 1 to represent the corresponding community. Each file is coupled with a mask that represents the communities that either have interacted with the file or have exchanged some information through the file. Anytime a user attempts an operation on a file, the module computes an `OR` of the masks of the file and of the user community. If the result shows an illegal information flow among communities, then the operation is forbidden, otherwise the result becomes the new mask of the file, in the case of a write operation, or of the community, in the case of a read operation. In general, files and communities may have more than one bit set to 1, which shows the interaction among communities. In any case, the Tainting module logs into a file the operation type, the name of the community and of the file and the original and the new masks. To this purpose, we have modified the `nfsd_permission` function, which verifies file requests, and `nfsd_vfs_read` and `nfsd_vfs_write` to check, respectively, read and write requests.

Periodically, the Tainting module parses the log file and updates in an incremental way a dependency graph [136] that represents the information flows among communities and files. Each node in the graph represents either a file or a community and it is coupled with a unique identifier of the community or of the file as well as with the corresponding mask. A node that represents a file is created the first time the file is involved in an operation. An arc represents an operation and is associated with the information about the requested operation. To identify information flows, a read operation is represented by an arc from a node that represents a file to one that represents a community, while a write operation by an arc from a community to a file. As shown in Fig. 8.4, an administrator can query the module to analyze the dependency graph to discover those communities that have exchanged some information, to trace the source of a contamination and track all the files/communities that may have been contaminated by a community/file.

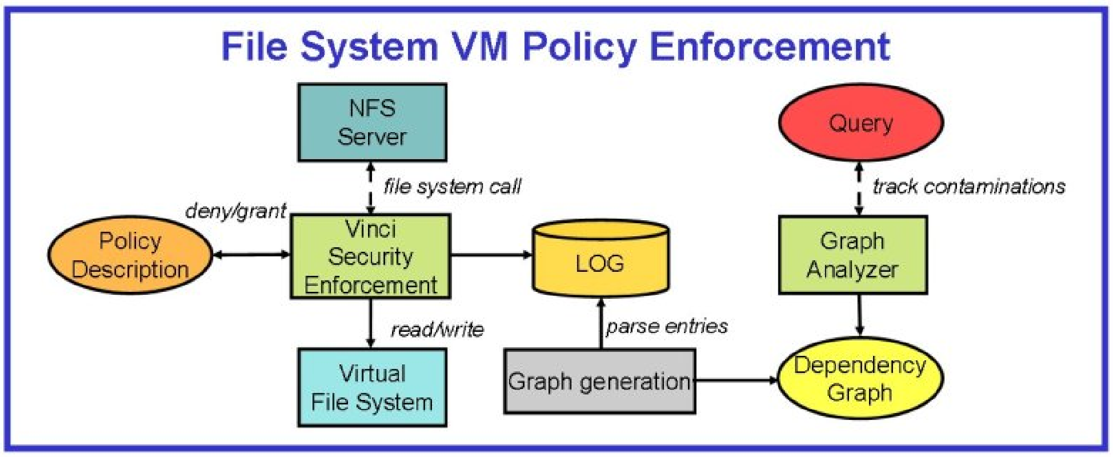


Figure 8.4: File System Virtual Machine Policy Enforcement and Query Generation

8.2.4 Communication and Control Virtual Machines

A CC-VM protects and monitors information flows by implementing and managing a local virtual switch that supports communications among VMs. CC-VMs can be further specialized into:

1. *VPN VMs*, which create an authenticated and protected communication channel among VMs of the same community mapped onto distinct physical nodes;

2. *Firewall VMs*, which filter information flows so that only authorized APP-VMs can access the physical or virtual communication components of the infrastructure and interact with other communities;

3. *IDS VMs*, which monitor information flowing among VMs in the same community or in distinct ones.

The introduction of a Firewall VM enables the infrastructure administrators to define which communities can interact and, hence, how VCNs can be connected into larger overlays. As an example, a community can be isolated or communities with a lower global level may not be allowed to communicate with those with higher global levels. Firewall VMs can decide whether to forward a packet, according to the source and destination communities. Furthermore, Firewall VMs support the authentication of shared resource requests through the IP address of the originating VM because they control that APP-VMs do not spoof traffic on the virtual switches interconnecting the VMs. Finally, IDS VMs monitor information flows across the same or distinct VCNs to discover attacks. An IDS VM can also retrieve and correlate partial information from other IDS VMs in the same VCN or in distinct ones to minimize the time to detect a distributed attack.

8.2.5 Assurance Virtual Machines

As described in Sect. 6.1.2, A-VMs can fully exploit PsycoTrace tools to monitor critical APP-VMs and the VMs of a VCN that manage critical components, to attest their integrity and to authenticate their configuration as well.

8.2.6 Infrastructure Virtual Machines

INF-VMs extend the VMMs to configure and manage the VCNs. In particular, distinct INF-VMs cooperate to monitor the overall infrastructure and update the topology of the VCNs and their mapping onto the physical architecture. The introduction of these VMs minimizes the size of the VMM by simplifying the implementation of some functionalities too complex to be developed at the VMM level. An INF-VM runs a minimal kernel, it does not run any Internet service and the functionalities it implements cannot be directly accessed by any user but the administrators.

As shown in Fig. 8.5, all the INF-VMs, one per node, belong to a *Management Community* that does not interact with any other community in a direct way since any interaction results in an update of the management of the infrastructure. During the creation of the Management Community, one INF-VM is designated as the

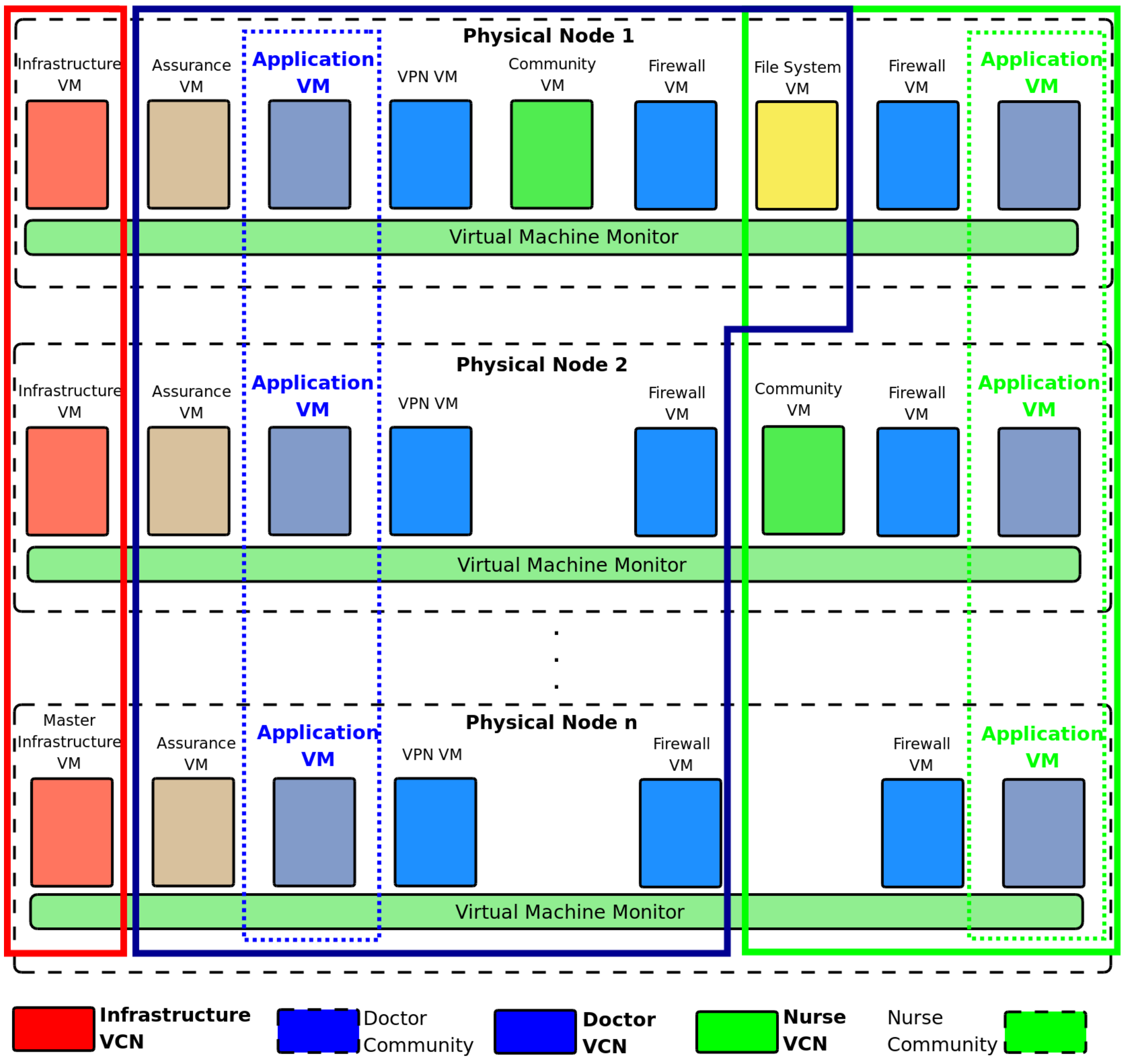


Figure 8.5: Example of Communities and Virtual Community Networks

Master INF-VM that contacts the other INF-VMs to set up proper communication channels to support cooperation in the Management Community.

To properly configure the Vinci run-time environment, INF-VMs can:

- create/kill, freeze/resume any VM in their node or request this operation to another INF-VM;
- configure a VM through specific parameters such as network configuration, amount of memory, the number of VCPUs;
- retrieve information about the current mapping of VMs and the resulting resource usage;

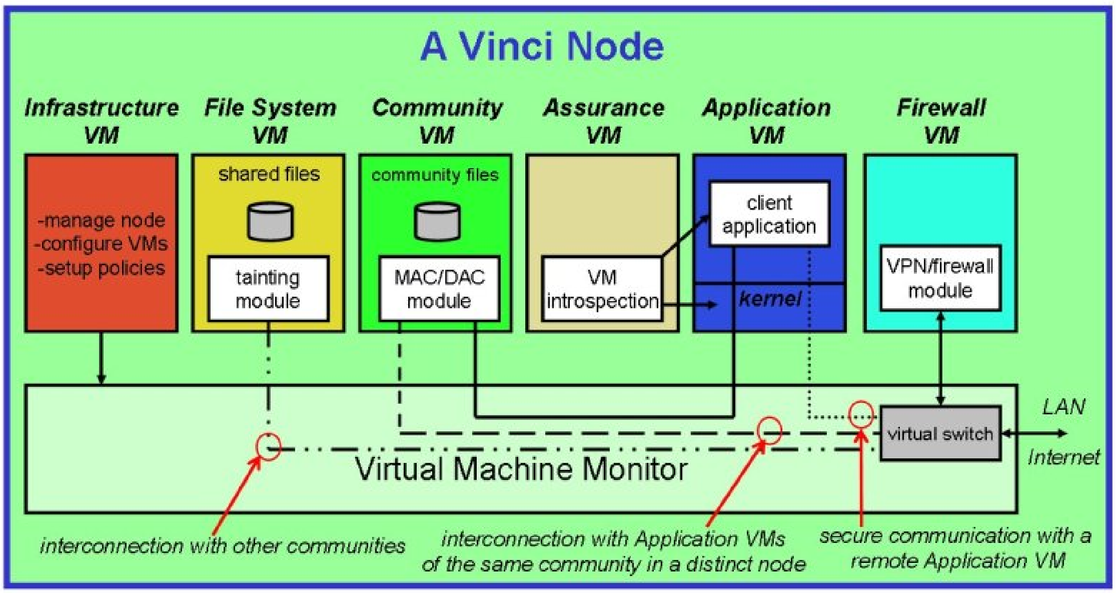


Figure 8.6: A Virtual Interacting Network Community Node

- update the mapping by migrating VMs, which requires an interaction with some CC-VMs to manage the resulting communications;
- setup, compile and deliver to each File System and COM-VM the general security policy it has to enforce.

Among the challenges that the Management Community has to face, one is concerned with data management issues, to enable a fast access of a community to its data [83], or with VMs mapping. Alternative mapping strategies may evenly distribute APP-VMs running server applications on the available nodes, or map COM-VMs onto physical nodes directly connected to those that run the APP-VMs of the corresponding community. The Management Community may migrate VMs among physical nodes to handle errors and faults, to reduce the communication latency or to balance the computational load.

INF-VMs also authenticate users through a centralized authentication protocol, so that users can log on APP-VMs with the same combination of user-name and password anywhere in the infrastructure. In this way, the association among users and privileges is managed in a centralized way. The set of users that share an infrastructure is globally known so that Vinci can uniquely identify users through their user-name or their associated user identifier (UID). The UID is coupled with the privileges of the user, i.e. with its security level. Whenever a user has been authenticated and has chosen the community she wants to join, the INF-VM starts up an APP-VM, which runs only those applications that satisfy the community policy and with the proper global level. After the login and boot-up phases, the local INF-VM contacts the proper CC-VMs to update the topology of the VCN to insert this APP-VM into a VCN and to add communication rules to handle the corresponding information flows. Finally, the security policies of COM-VMs and FS-VMs may be updated. Figure 8.6 shows the various interactions among the VMs running on a physical node. Currently, INF-VMs are assigned to Xen Domain 0 VMs, i.e. they are privileged VMs that can access the control interface to manage a physical node and the VMs that the node runs.

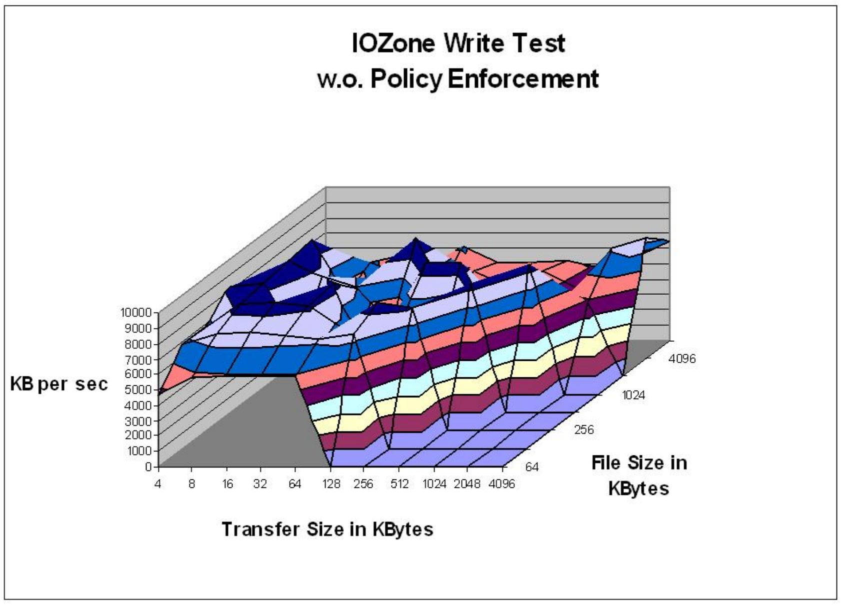


Figure 8.7: IOzone NFS Read Performance without Policy Enforcement

8.3 Performance Results

This section shows a preliminary performance evaluation of the current Vinci prototype. The tests were performed on several machines equipped with Intel Core 2 Duo E6550 2.33GHz CPUs. A first experiment evaluated in an integrated way the performance of file sharing through FS-VMs and CC-VMs. An APP-VM on a node ran the IOzone [120] NFS test while a FS-VM, on a distinct physical node, stored the requested files. Requests were transmitted along a communication channel implemented by two Firewall VMs and the physical nodes were connected through 100MB Ethernet. Figures 8.7 and 8.8 compare the throughput of the `write` test against the one of the insecure version that does not apply the security checks. The overhead due to the enforcement of the security policies, in the average case, is lower than 9%. Instead, the tests on the enforcement of security policies by a COM-VM resulted in a reduction of the final throughput lower than 5%. The same tests executed on an APP-VM connected to a remote FS-VM through two VPN VMs resulted in an overhead that, in the worst case, is lower than 13%.

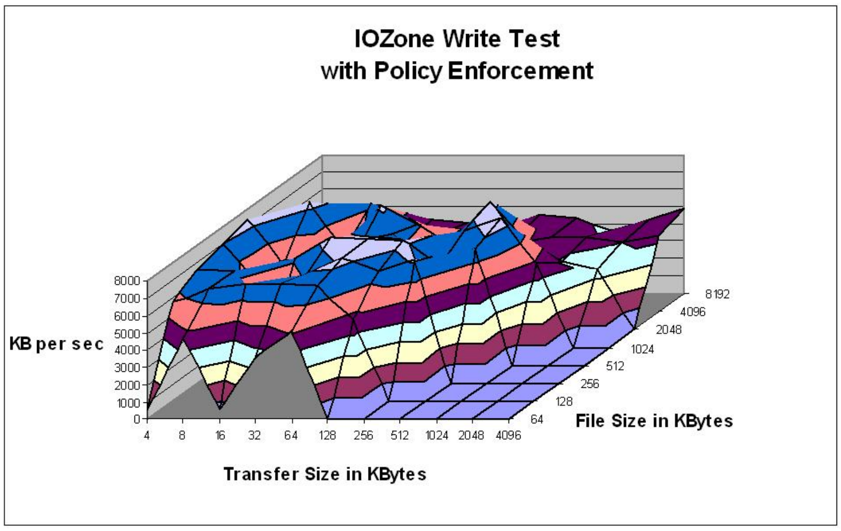


Figure 8.8: IOzone NFS Read Performance with Policy Enforcement

Part IV

Final Remarks

Conclusions

In this thesis we have discussed PsycoTrace, a virtualization-based framework to protect the integrity of a process self by detecting and preventing most attacks that modify the intended behavior of the process. The key feature of the framework is the integration of static analysis, which returns a specification of the process self that corresponds to the expected behavior of the process, and run-time monitoring, which compares the actual behavior of the process against the specifications to discover any deviation. PsycoTrace supports alternative specifications of the self where the most severe strategy consists of a context-free grammar that describes sequences of system calls issued by the process and a set of assertions to be evaluated when a system call is executed. The grammar and the assertions are the output of the static analysis that exploits the notion of control-flow graph and introduces the concept of system block, i.e. the sequence of instructions in-between two consecutive system calls. At run-time, each time the process invokes a system call, PsycoTrace tools apply introspection to access the state of the monitored process and verify that the system call trace is coherent with the grammar and that the assertion coupled with the system call is verified. If a system call trace is incompatible with the grammar, or an assertion is false, PsycoTrace assumes that the process has been successfully attacked.

Virtualization technology enables us to implement the proposed framework without introducing specialized hardware/firmware units. Furthermore, it helps us to build a robust monitoring system by running two distinct virtual machines: the first one executes the monitored system while the second one exploits virtual machine introspection to check the process self. Virtual machine introspection also enables the protection of kernel integrity by monitoring both its code and critical data structures from illegal updates, thus assuring that the trusted computing base has not been compromised. A fundamental advantage of PsycoTrace is transparency because its adoption does not require to modify the programs and the components to be protected.

One of the problems posed by introspection is the semantic gap between the raw access to data and the abstraction level to define invariants to detect attacks at the OS-level. To deal with the semantic-gap problem, we have extended the framework with the introduction of a context-agent, i.e. a program fragment that is injected into the monitored virtual machine in an almost total transparent way to complement the low-level view that the introspection virtual machine has of the monitored virtual machine. The context-agent implements short-term living services that run inside the monitored virtual machine to: (i) return fresh information on the running state of the virtual machine; (ii) cooperate with the introspection virtual machine to detect inconsistencies in the kernel.

We have also discussed the applications of the framework to three fields, namely (i) remote attestation of semantic integrity; (ii) code obfuscation to protect the virtual machines from physical attacks; (iii) highly parallel and trusted overlays. The application of the framework to remotely attest the semantic-integrity of a system can overcome the limitations of the Trusted Computing framework. Code obfuscation is implemented by splitting the program in a control-part, located in a distinct virtual machine, and a further program composed of a set of encrypted system blocks. Finally, the adoption of PsycoTrace to define highly parallel overlays of virtual machines has led us to the definition of Vinci, which aims to simplify the management of an ICT infrastructure.

Future Works

We can partition the work to deepen and evaluate the ideas in this thesis along two main aspects. The first one concerns PsycoTrace framework itself, the other one the applications of PsycoTrace that we have discussed. As far as concerns the first aspect, we believe that a more formal description of the static analysis and of the strategies to describe the process self may result in a more accurate description of the expected behavior and in a lower number of false negatives, due to the lower number of undetected attacks, e.g. mimicry attacks. A further interesting development is the adoption of an asynchronous cooperation model among the virtual machines in the run-time support. Another important extension concerns the several security problems that PsycoTrace does not cover, such as time of check time of use errors [33]. Moreover, there are several non-standard control-flows that a static analysis cannot handle very easily. As an example, a function pointer could be used to indirectly invoke a system call. To take into account function pointers, the static analysis should predict all the possible targets of every indirect call through a function pointer. For the moment being, we neglect function pointers, so our approach may miss some system call invocation and we neglect linked libraries as well. In this case, if the source code of the library is available, we can apply the static analysis, otherwise we can associate each function of the library with a context-free grammar only by reverse engineering the library. The handling of dynamic linking is even more complex, because a process can load at run-time any library. This requires to dynamically update both $CFG(P)$ and the $IT(P)$, each time a new library is loaded. An alternative solution inserts *null calls* into the program code to signal the I-VM that a library function has been invoked to stop/resume the checks on the Mon-VM. Further non-standard control-flows are direct invocations of system calls, e.g. using inlined assembly instructions, that should also be located and correctly decoded. Finally, another non-standard control-flow mechanism is the OS signal facility. Since a process may receive signals in any order and the handler of a signal may invoke several system calls, a static analysis cannot extract their order in $CFG(P)$.

With respect to the applications of PsycoTrace discussed in this thesis, first of all some experiences with a real-world context would be very useful. From this point of view, a critical problem to be deepened is the reduction of the virtualization overhead by properly exploiting the extension of physical processors to efficiently support virtualization technology[152]. A future development of VIMS considers the exploitation of an USB dongle as a secure root-of-trust of the VMM and the A-VM to increase the portability of this architecture to those contexts where the adoption of a TPM chip gives rise to privacy concerns. An important improvement of the obfuscation strategy is related to virtual memory. In fact, at run-time, some of the blocks may not have been loaded into memory, because of the paging mechanism. This problem can be solved by introducing a trigger mechanism [242] so that the I-VM starts decrypting the system block as soon as it is going to be executed in memory. Moreover, the solutions previously discussed as far as concerns linked libraries may be applied in this case as well. A counterpart of the advantages of virtual, highly parallel and secure overlays is the overhead due to the context switching that the VMM applies to multiplex the physical resources. A multi-core architecture [97] can strongly reduce this overhead because of the native support for multiplexing. Moreover, this architecture can run several VMs in consolidation and assign a dedicated core to some VMs. This guarantees that VMs that implement critical tasks, such as management and protection of other VMs, are never delayed.

List of Acronyms

APP-VM	Application Virtual Machine
A-VM	Assurance Virtual Machine
A-VM$_{ove}$	The A-VM running on N_{ove}
A-VM$_{req}$	The A-VM that Checks Mon-VM$_{req}$
CC-VM	Communication and Control Virtual Machine
$CFGraph(P)$	The Control-Flow Graph of P
$CFG(P)$	The Context-Free Grammar of P
COM-VM	Community Virtual Machine
FSC	File Sharing Client
FSS	File Sharing Server
FS-VM	File System Virtual Machine
GGA	Grammar Generating Algorithm
INF-VM	Infrastructure Virtual Machine
$IT(P)$	Invariant Table for P
I-VM	Introspection Virtual Machine
$L(P)$	The Language Generated by $CFG(P)$
MAC-S	Mandatory Access Control-based Security Module
Mon-VM	Monitored Virtual Machine
Mon-VM$_{req}$	The Mon-VM running on N_{req}

ove	A Generic Overlay
N_{ove}	A Node of the Overlay ove
N_{req}	A Node that Requests Access to the Overlay ove
P	The Process to be Protected
$Self(P)$	The Process Self of P
$SourceCode(P)$	The Source Code of the Program Executed by P
SB	System Block
SBG	System Block Graph
STO-VM	Storage Virtual Machine
UB	Unit Block
UBG	Unit Block Graph
VCN	Virtual Community Network
VIMS	Virtual machine Integrity Measurement System
VINCI	Virtual Interacting Network CommunIty
VM	Virtual Machine
VMI	Virtual Machine Introspection
VMM	Virtual Machine Monitor

Appendix A

Meta-Compiler-Compiler Approach

Currently, GGA is implemented by a Java program that also generates the program's invariants by traversing $AST(P)$. An alternative implementation that we have developed builds $CFG(P)$ by exploiting Bison to generate a parser for an extended version of the C language where system calls are tokens of the language. This parser implements GGA and its semantic actions generate $CFG(P)$. Moreover, this solutions also exploits Bison to build a second parser that at run-time checks if the trace of P is a a prefix of at least one string of $L(P)$. In more detail, this *meta-compiler-compiler approach* (see Fig. A.1) is implemented in three steps, where the first step does not depend upon $SourceCode(P)$ and each of the remaining steps builds a distinct parser:

1. define an extended C grammar (ECG) in the Bison syntax where system calls are added as new tokens. We also define the semantic actions coupled with the rules of ECG that implement GGA to generate $CFG(P)$;

2. apply Bison to ECG to produce the parser that generates $CFG(P)$ when applied to $SourceCode(P)$. Semantic actions for $CFG(P)$ return the assertion that holds at each system call invocation;

3. apply Bison to $CFG(P)$ to build the parser that checks if the current trace of P is a prefix of a string in $CFG(P)$. The semantic actions associated with this parser include the evaluation of assertions.

In this solution, in step (1) a Flex-generated scanner recognizes system calls as tokens of the language. To create $CFG(P)$, the parsing of $SourceCode(P)$ in step (2) is decomposed into two further steps. In the first step, the semantic actions of the generated parser build and export $AST(P)$. In the second step, the parser visits the $AST(P)$ to build $CFG(P)$ by applying GGA. $CFG(P)$ is represented in the Bison syntax so that in step (3) we can exploit Bison to generate the parser to check the current trace of P. The implementation of the ECG, which also includes the semantic actions to generate $CFG(P)$, is about 1K lines of C++ code.

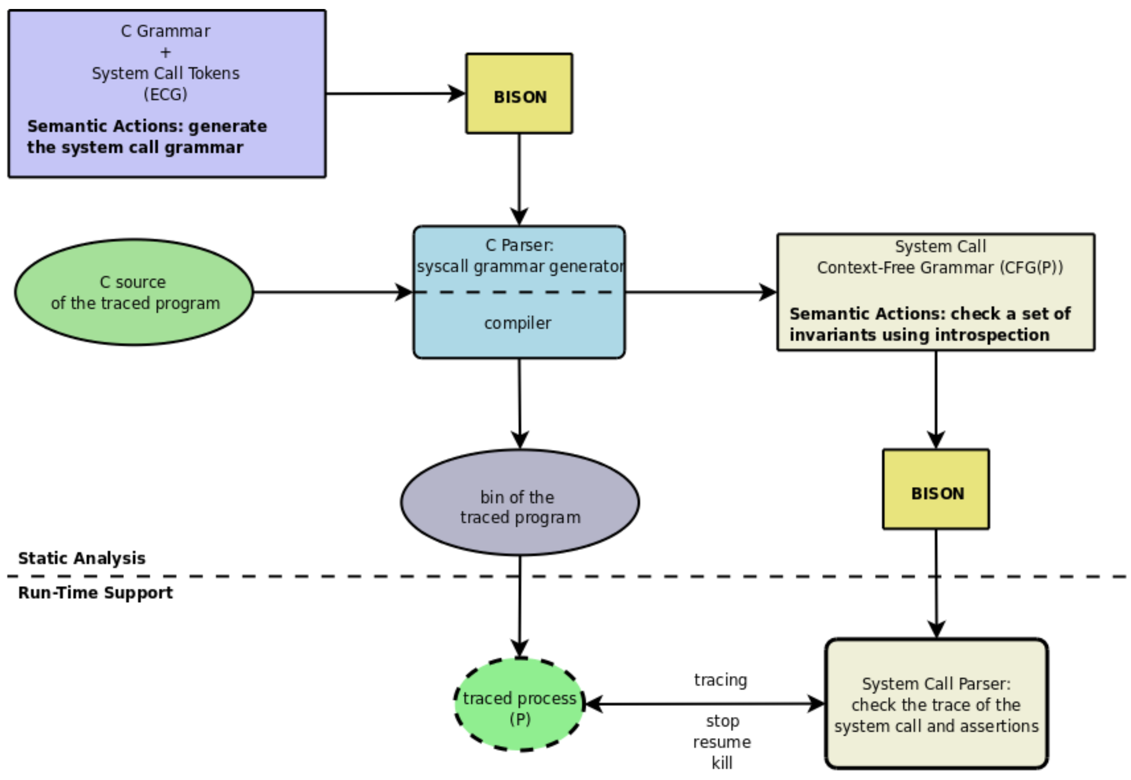

C Grammar
+
System Call Tokens
(ECG)
Semantic Actions: generate the system call grammar
BISON
C source of the traced program
C Parser: syscall grammar generator
compiler
System Call Context-Free Grammar (CFG(P))
Semantic Actions: check a set of invariants using introspection
BISON
bin of the traced program
Static Analysis
Run-Time Support
traced process (P)
tracing
stop
resume
kill
System Call Parser: check the trace of the system call and assertions

Figure A.1: Meta-Compiler-Compiler Approach

Appendix B

Retrieving Data-Structures through the Introspection Library

This appendix describes in details the algorithms that the I-VM implements to retrieve and rebuilds some kernel data-structures in the Mon-VM memory.

Retrieving the Page Global Directory Address. The I-VM needs to retrieve the Page Global Directory (PGD) address so that the Introspection Library can convert virtual addresses into physical ones. We have adopted an approach where the I-VM retrieves automatically the PCB of each process in the Mon-VM by rebuilding and traversing the running processes list until it finds the `init` PCB and from this PCB it retrieves the corresponding PGD.

To retrieve the PGD address of the `init` process, the I-VM traverses the Mon-VM's list of running processes, it finds the corresponding PCB, a struct `task_struct` in Linux, that stores the PID 1 and it retrieves the corresponding PGD. The PGD is a field of `mm` (a `struct mm_struct`), a field of the `task_struct`. The I-VM executes the following algorithm to locate the PGD value inside a PCB (see Fig. B.1):

1. retrieve the first element of the Mon-VM running processes list, pointed by the kernel symbol `init_task`;
2. invoke an introspection function to:
   (a) cast to a `task_struct` the buffer that stores the memory of the current task;
   (b) retrieve the address of the next process in the field: this is implemented through `next_task()`, a macro defined in `linux/sched.h`, which returns the next task in the list.
3. invoke an introspection function that retrieves the PID value of the current task. The process list is scanned till reaching the hijacked process;

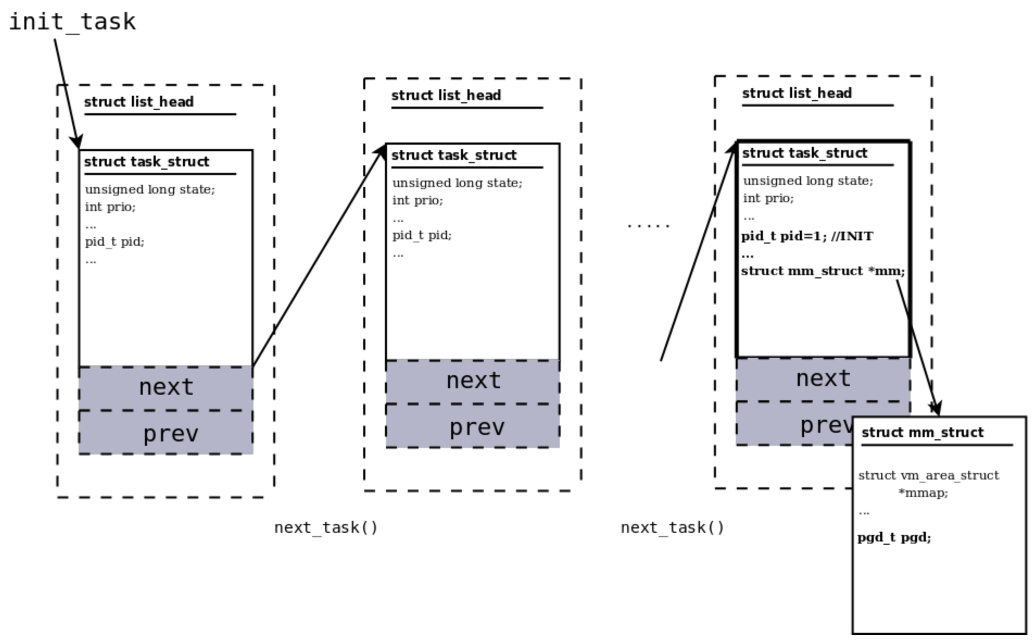


Figure B.1: Retrieving the Page Global Directory through the Introspection Library

4. retrieve the linear address of the `mm` field, which is a data structure that contains, among the others, the PGD value, and translate it into a physical one;

5. retrieve the content of the address of the `mm` field;

6. cast the buffer to a `struct mm_struct *`, and retrieve the value of the field `PGD` in this structure.

Anytime the I-VM invokes the Introspection Library to rebuild a kernel data-structure, the library casts the region of memory to the corresponding data-structure declared in the Linux headers. To this end, the I-VM executes the following algorithm:

1. the I-VM retrieves a page of the Mon-VM's memory containing a kernel data structures, such as `modules` and `init_task`. This is implemented by invoking an introspection function to read the physical pages of a Mon-VM [1];

2. an introspection function casts the memory to the correct data structures, such as `struct task_struct *` or `struct module *`. To implement the casting correctly, the I-VM requires the header files of the Mon-VM's Linux kernel that include the declaration of the considered data structures;

[1]In the current prototype, the linear addresses of the kernel symbols are retrieved from the corresponding Mon-VM's `System.map` files: since these variables can be reached by some system calls, their addresses can be found by scanning the region of memory that can be reached by such system calls and looking for known signatures.

3. if a kernel data structure contains a pointer (such as the field `mm` of the `struct task_struct *`), an introspection function returns the value of the linear address of this pointer and the I-VM converts it into a physical one. Then, it retrieves the page that contains this address and casts the pointed structure to the correct kernel data structure.

Table B.1 shows the code to retrieve the list of running processes in Mon-VM OS that exploits the previous approach.

Retrieving the Context-Agent. The I-VM also retrieves the address of the *context-agent* and its code from the Mon-VM's memory to compute the hash of the code to verify its integrity. To protect the integrity of the *context-agent*, the I-VM has to retrieve the kernel pages storing the *context-agent* and compute their hash. The I-VM applies the following algorithm to retrieve the code of the *context-agent* from the Mon-VM's memory:

1. get the linear address of the first module in the list of the loaded kernel modules; then cast the memory to a `struct list_head *` structure, which is the one pointed by the `modules` kernel symbol. Then, the I-VM retrieves the `module struct` using the Linux macro `list_entry()`. Since the first entry is not a module entry, but simply the head of the list, it is handled differently than the other ones. For this reason, the I-VM needs to invoke the Linux macro `next_module()` to retrieve the address of the first module;

2. from this moment on, the I-VM exploits an introspection function to retrieve the linear address of the next module, which then it translates to the physical one;

3. the memory storing the `module struct` is retrieved, and an introspection function is invoked to get and compare the name of the module in the list against the one of the *context-agent*;

    - if the name of the module in the list is the same of the *context-agent*, the I-VM retrieves from the corresponding `module struct` the address and size of its code and finally I-VM retrieves the *context-agent*'s code;
    - otherwise, the I-VM retrieves both the address of the next kernel module and that of the node in the list storing the next kernel module, so that it can compare this address against the first one of the list. If the two addresses differ, then the I-VM translates the linear address of the next module, and retrieves the memory storing its information and so on until it locates the injected kernel module's name.

```
int get_process_list(uint32_t vm)
{
     struct task_struct *task;            //pointer to the current process
     uint32_t init_addr, next_addr;       //address of the first and next process
     char *symbol = "init_task";          //kernel name of the first process
     void *mem_mon_vm;                    //pointer to the Mon-VM memory
     int xc_handle = xc_interface_open(); //open the VMM control interface

     //retrieve the kernel address of the list from System.map file
     init_addr = get_kernel_address(symbol);
     [...]

     //1)PAUSE THE MON-VM
     xc_vm_pause(xc_handle, vm);

     //2)GET THE FIRST PCB OF THE LIST
     mem_mon_vm = get_mem_mon_vm(xc_handle, 0, vm, (void *)init_addr,
                                 PROT_READ | PROT_WRITE, KERNEL_PGD);
     [...]

     while(next_addr != init_addr) //enumerate all the processes
     {
          //3)CAST THE MEMORY TO THE PCB KERNEL DATA STRUCTURE
          task = (struct task_struct *)mem_mon_vm;

          //4)RETRIEVE THE ADDRESS OF THE NEXT PCB IN THE LIST
          next_addr = (uint32_t)next_task(task);
          [...]

          //5)RETRIEVE THE NEXT PCB
          mem_mon_vm = get_mem_mon_vm(xc_handle, 0, vm, (void *)next_addr,
                                      PROT_READ | PROT_WRITE, KERNEL_PGD);
          //6)APPLY CHECK ON THE CURRENT PROCESS
          [...]
     }
     //7)RESUME THE MON-VM
     xc_vm_unpause(xc_handle, vm);
     free(mem_mon_vm);
     xc_interface_close(xc_handle);
     return 0;
}
```

Table B.1: Retrieving the Process List of the Monitored Virtual Machine

Appendix C

Source Code of the Testbed Program

```c
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <netdb.h>
#include <unistd.h>

#define BUFF 1024
#define SMALL_BUFF 512

int n = 10;

void logfile()
{
   int fd = open(``log.txt'', O_CREAT|O_WRONLY|O_APPEND, S_IRWXU);
   n--;
   write(fd, ``log'', 3);
   if(n>0) logfile();
   else (n=10);
   close(fd);
}

int parse_str(char *buff)
{
   char smallbuff[SMALL_BUFF];
   if(!strncmp(buff, ``copy'', 4)) strcpy(smallbuff, buff); /* VULNERABILITY*/
   if(!strncmp(buff, ``file'', 4)) logfile();
   if(!strncmp(buff, ``exit'', 4)) return 1;
   return 0;
}

int main()
{
   int fd, sockfd, ret, yes=1;
   socklen_t sin_size;
   struct sockaddr_in sin;
   struct hostent *h;
   char buffer[BUFF];
   fd = socket(AF_INET, SOCK_STREAM, 0);
   memset(&sin, 0, sizeof(sin));
   h = gethostbyname(``localhost'');
   sin.sin_family = AF_INET;
   sin.sin_port = htons(5555);
   sin.sin_addr.s_addr = INADDR_ANY;
   ret = setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
   ret = bind(fd, (struct sockaddr *) &sin, sizeof(sin));
   ret = listen(fd, 5);
   sin_size = sizeof(struct sockaddr_in);
   sockfd = accept(fd, (struct sockaddr *)&sin, &sin_size);
   dup2(sockfd, STDIN_FILENO);
   dup2(sockfd, STDOUT_FILENO);
   dup2(sockfd, STDERR_FILENO);
   while(1)
   {
      memset(buffer, 0, BUFF);
      read(sockfd, buffer, BUFF);
      if(parse_str(buffer) == 1) break;
      write(sockfd, ``ok\n'', 3);
   }
   close(sockfd);
}
```

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. A theory of secure control flow. *Lecture notes in computer science*, 3785:111, 2005. 37

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353, New York, NY, USA, 2005. ACM. 37

[3] Charles Reis Adam Barth, Collin Jackson and Google Chrome Team. The security architecture of the Chromium browser. Technical report, Stanford University, 2008. xviii

[4] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996. xvi

[5] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002. xx

[6] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. Proteus: virtualization for diversified tamper-resistance. In *DRM '06: Proceedings of the ACM workshop on Digital rights management*, pages 47–58, New York, NY, USA, 2006. ACM. 52

[7] J.P. Anderson et al. Computer Security Technology Planning Study. Technical report, ESD-TR-73-51, 1972. xvi

[8] Melvin J. Anderson, Micha Moffie, and Chris I. Dalton. Towards Trustworthy Virtualisation Environments: Xen Library OS Security Service Infrastructure. Technical Report HPL-2007-69, Hewlett-Packard Laboratories, April 2007. 54

[9] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 65, Washington, DC, USA, 1997. IEEE Computer Society. 49

[10] Darren C. Atkinson and William G. Griswold. Implementation Techniques for Efficient Data-Flow Analysis of Large Programs. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 52–61, Washington, DC, USA, 2001. IEEE Computer Society. 61

[11] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Trans. Inf. Syst. Secur.*, 3(3):186–205, 2000. xix

[12] Fabrizio Baiardi, Diego Cilea, Daniele Sgandurra, and Francesco Ceccarelli. Measuring Semantic Integrity for Remote Attestation. In *Trusted Computing, Second International Conference, Trust 2009, Oxford, UK, April 6-8, 2009, Proceedings*, volume 5471 of *Lecture Notes in Computer Science*, pages 81–100. Springer, 2009. xxiv

[13] Fabrizio Baiardi, Dario Maggiari, and Daniele Sgandurra. Invariant Evaluation through Introspection for Proving Security Properties. *Information Assurance and Security*, 4(2):124–132, 2009. xxiv

[14] Fabrizio Baiardi, Dario Maggiari, and Daniele Sgandurra. Securing Health Information Infrastructures through Overlays. In Luís Azevedo and Ana Rita Londral, editor, *Proceedings of the Second International Conference on Health Informatics, HEALTHINF 2009, Porto, Portugal, January 14-17, 2009*, pages 123–128. INSTICC Press, 2009. xxiv

[15] Fabrizio Baiardi, Dario Maggiari, Daniele Sgandurra, and Francesco Tamberi. PsycoTrace: Virtual and Transparent Monitoring of a Process Self. In Didier El Baz, François Spies, and Tom Gross, editors, *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009, Weimar, Germany, 18-20 Febuary 2009*, pages 393–397. IEEE Computer Society, 2009. xxiii

[16] Fabrizio Baiardi, Dario Maggiari, Daniele Sgandurra, and Francesco Tamberi. Transparent Process Monitoring in a Virtual Environment. *Electronic Notes in Theoretical Computer Science*, 236:85 – 100, 2009. Proceedings of the 3rd International Workshop on Views On Designing Complex Architectures (VODCA 2008). xxiii

[17] Fabrizio Baiardi, Gaspare Sala, and Daniele Sgandurra. Managing Critical Infrastructures through Virtual Network Communities. In *2nd IEEE-IFIP International Workshop on Critical Information Infrastructures Security (CRITIS '07)*, volume 5141 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 2007. xxiv

[18] Fabrizio Baiardi and Daniele Sgandurra. Building Trustworthy Intrusion Detection through VM Introspection. In *IAS '07: Proceedings of the Third International Symposium on Information Assurance and Security*, pages 209–214, Washington, DC, USA, 2007. IEEE Computer Society. xxiv

[19] Fabrizio Baiardi and Daniele Sgandurra. Towards High Assurance Networks of Virtual Machines. In *Proc. of 3rd European Conference on Computer Network Defense (EC2ND)*, volume 30 of *Lecture Notes in Electrical Engineering*, pages 21–34, Heraklion, Greece, October 2007. xxiv

[20] Fabrizio Baiardi and Daniele Sgandurra. Secure Sharing of an ICT Infrastructure through Vinci. In David Hausheer and Jürgen Schönwälder, editors, *Resilient Networks and Services, Second International Conference on Autonomous Infrastructure,*

Management and Security, AIMS 2008, Bremen, Germany, July 1-3, 2008, Proceedings, volume 5127 of *Lecture Notes in Computer Science*, pages 65–78. Springer, 2008. xxiv

[21] Fabrizio Baiardi and Daniele Sgandurra. Virtual Interacting Network Community: Exploiting Multi-core Architectures to Increase Security. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 111–112, New York, NY, USA, 2008. ACM. xxiv

[22] S. Bajikar. Trusted Platform Module (TPM) based Security on Notebook PCs-White Paper. *Mobile Platforms Group, Intel Corporation, June*, 20, 2002. 49, 110

[23] A. Baliga, X. Chen, and L. Iftode. Paladin: Automated detection and containment of rootkit attacks. Technical Report DCS-TR-593, Department of Computer Science, Rutgers University, April 2006. 43

[24] Mohammad Banikazemi, Dan Poff, and Bulent Abali. Storage-based file system integrity checker. In *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 57–63, New York, NY, USA, 2005. ACM Press. 57

[25] Mohammad Banikazemi, Dan Poff, and Bulent Abali. Storage-Based Intrusion Detection for Storage Area Networks (SANs). In *MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 118–127, Washington, DC, USA, 2005. IEEE Computer Society. 56

[26] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM. 16, 78, 117, 154

[27] Massimo Bartoletti, Gabriele Costa, Pierpaolo Degano, Fabio Martinelli, and Roberto Zunino. Securing Java with Local Policies. *Journal of Object Technology*, 8(4):5–32, 2009. 38

[28] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. Model Checking Usage Policies. In *Trustworthy Global Computing, 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers*, volume 5474 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2009. 38

[29] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: virtualizing the trusted platform module. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2006. USENIX Association. 51, 111

[30] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. Operating system enhancements to prevent the misuse of system calls. In *CCS '00: Proceedings of the 7th ACM conference on Computer and communications security*, pages 174–183, New York, NY, USA, 2000. ACM. 95

[31] Sandeep Bhatkar, Abhishek Chaturvedi, and R. Sekar. Dataflow Anomaly Detection. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 48–62, Washington, DC, USA, 2006. IEEE Computer Society. 41

[32] P. Biondi and F. Desclaux. Silver needle in the Skype. *BlackHat Europe*, 6, 2006. 52

[33] M. Bishop and M. Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 2(2):131–152, 1996. 172

[34] Kevin Borders, Xin Zhao, and Atul Prakash. Securing sensitive content in a view-only file system. In *DRM '06: Proceedings of the ACM workshop on Digital rights management*, pages 27–36, New York, NY, USA, 2006. ACM Press. 57

[35] R. Bradshaw, N. Desai, T. Freeman, and K. Keahey. A Scalable Approach To Deploying And Managing Appliances. In *TeraGrid 2007*, pages 1–6, June 2007. 55

[36] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. An Efficient Technique for Preventing Mimicry and Impossible Paths Execution Attacks. *Performance, Computing, and Communications Conference, 2002. 21st IEEE International*, 0:418–425, 2007. 40

[37] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Static Analysis on x86 Executables for Preventing Automatic Mimicry Attacks. In *DIMVA '07: Proceedings of the 4th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 213–230, Berlin, Heidelberg, 2007. Springer-Verlag. 40

[38] Bryan D. Payne and Martim Carbone and Wenke Lee. Secure and flexible monitoring of virtual machines. *Computer Security Applications Conference, Annual*, 0:385–397, 2007. 44

[39] E. Bryant, J. Early, R. Gopalakrishna, G. Roth, EH Spafford, K. Watson, P. William, and S. Yost. Poly^2 Paradigm: A Secure Network Service Architecture. *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 342–351, 2003. 53

[40] Serdar Cabuk, Chris I. Dalton, HariGovind Ramasamy, and Matthias Schunter. Towards automated provisioning of secure virtualized networks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 235–245, New York, NY, USA, 2007. ACM. 54

[41] Calvin Ko and George Fink and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, Orlando, FL, 1994. IEEE Computer Society Press. 35

[42] Martim Carbone, Diego Zamboni, and Wenke Lee. Taming Virtualization. *IEEE Security and Privacy*, 6(1):65–67, 2008. 47

[43] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 11–11, Berkeley, CA, USA, 2006. USENIX Association. 41

[44] Edjozane Cavalcanti, Leonardo Assis, Matheus Gaudencio, Walfredo Cirne, and Francisco Brasileiro. Sandboxing for a free-to-join grid with support for secure site-wide storage area. In *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 11, Washington, DC, USA, 2006. IEEE Computer Society. 54

[45] Abhishek Chaturvedi, Sandeep Bhatkar, Eep Bhatkar, and R. Sekar. Improving Attack Detection in Host-Based IDS by Learning Properties of System Call Arguments. Technical Report SECLAB-05-03, Department of Computer Science, Stony Brook University, Stony Brook, NY 11794, 2005. 35, 98

[46] L. Chen, R. Landfermann, H. Löhr, M. Rohe, A.R. Sadeghi, and C. Stüble. A protocol for property-based attestation. In *Proceedings of the first ACM workshop on Scalable trusted computing*, pages 7–16. ACM New York, NY, USA, 2006. 50

[47] Peter M. Chen and Brian D. Noble. When Virtual Is Better Than Real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, 2001. IEEE Computer Society. 28

[48] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 177–192, Berkeley, CA, USA, 2005. USENIX Association. 98

[49] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2008. ACM. 45

[50] Tzi cher Chiueh, Matthew Conover, Maohua Lu, and Bruce Montague. Stealthy Deployment and Execution of In-Guest Kernel Agents. In *Black Hat USA*, pages 1–12, 2009. 46

[51] Mihai Christodorescu, Reiner Sailer, Douglas Lee Schales, Daniele Sgandurra, and Diego Zamboni. Cloud security is not (just) virtualization security: a short paper. In *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 97–102, New York, NY, USA, 2009. ACM. 46

[52] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003. 52

[53] George Coker, Joshua Guttman, Peter Loscocco, Justin Sheehy, and Brian T. Sniffen. Attestation: Evidence and trust. In Liqun Chen, Mark Dermot Ryan, and Guilin Wang, editors, *ICICS*, volume 5308 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008. 115

[54] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures, October 1997. 52

[55] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, The University of Auckland, July 1997. 52

[56] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, January 1998. 52

[57] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on Software Engineering*, 28:735–746, 2002. 52

[58] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *2nd International Symposium on Programming*, pages 106–130, April 1976. xx

[59] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977. xx

[60] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, New York, NY, USA, 1978. ACM. xx

[61] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference & Exposition – Volume 2*, pages 119–129, Jan 2000. xv

[62] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998. xviii

[63] Alberto Daniel. Strategie di offuscamento del codice basate su macchine virtuali. Master's thesis, Università di Pisa, 2009. xxiv

[64] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Comput. Netw.*, 31(9):805–822, 1999. xviii

[65] Victor DeMarines. Obfuscation - how to do it and how to crack it. *Network Security*, 2008(7):4 – 7, 2008. 52

[66] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, New York, NY, USA, 2008. ACM. 49

[67] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 566–577, New York, NY, USA, 2009. ACM. 38

[68] C. Donnelly and R.M. Stallman. *The Bison manual: using the YACC-compatible parser generator*. Boston, MA: GNU, 2006. 7

[69] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 211–224, New York, NY, USA, 2002. ACM Press. 43

[70] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - Open Source Graph Drawing Tools. *Graph Drawing*, pages 483–484, 2001. 62

[71] W. Emeneker, D. Jackson, J. Butikofer, and D. Stanzione. Dynamic virtual clustering with Xen and Moab. *Proceedings of ISPA Workshops: Workshop on Xen in HPC Cluster and Grid Computing Environments (XHPC)*, pages 440–451, 2006. 54

[72] Paul England. Practical Techniques for Operating System Attestation. In *Trust '08: Proceedings of the 1st international conference on Trusted Computing and Trust in Information Technologies*, pages 1–13, Berlin, Heidelberg, 2008. Springer-Verlag. 51

[73] Paul England and Jork Loeser. Para-Virtualized TPM Sharing. In *Trust '08: Proceedings of the 1st international conference on Trusted Computing and Trust in Information Technologies*, pages 119–132, Berlin, Heidelberg, 2008. Springer-Verlag. 51

[74] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *International Conference on Software Engineering*, pages 213–224, 1999. xix

[75] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen Mccamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007. xx

[76] E. Eskin, Wenke Lee, and S.J. Stolfo. Modeling system calls for intrusion detection with dynamic window sizes. In *Proceedings of DARPA Information Survivability Conference & Exposition II, 2001. DISCEX '01*, volume 1, pages 165–175, 2001. 35

[77] D. Evans, J. Guttag, J. Horning, and Y.M. Tan. LCLint: a tool for using specifications to check code. *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 87–96, 1994. xix

[78] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly Detection Using Call Stack Information. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 62, Washington, DC, USA, 2003. IEEE Computer Society. 36

[79] H.H. Feng, J.T. Giffin, Yong Huang, S. Jha, Wenke Lee, and B.P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 194–208, May 2004. 36

[80] P. Ferrie. Attacks on Virtual Machine Emulators. Technical report, Symantec Security Response, December 2006. 48

[81] P. Ferrie. Attacks on More Virtual Machine Emulators. Technical report, Symantec Security Response, February 2007. 48

[82] Christof Fetzer and Martin Süsskraut. Switchblade: enforcing dynamic personalized system call models. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 273–286, New York, NY, USA, 2008. ACM. 38

[83] Renato J. Figueiredo, Peter A. Dinda, and José A. B. Fortes. A Case For Grid Computing On Virtual Machines. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 550, Washington, DC, USA, 2003. IEEE Computer Society. 166

[84] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. Joining dataflow with predicates. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 227–236, New York, NY, USA, 2005. ACM. 41

[85] Jason Franklin, Mark Luk, Jonathan M. McCune, Arvind Seshadri, Adrian Perrig, and Leendert van Doorn. Remote detection of virtual machine monitors with fuzzy benchmarking. *SIGOPS Oper. Syst. Rev.*, 42(3):83–92, 2008. 47

[86] Jason Franklin, Mark Luk, Jonathan M. McCune, Arvind Seshadri, Adrian Perrig, and Leendert van Doorn. Towards Sound Detection of Virtual Machines. In *Botnet Detection*, volume 36 of *Advances in Information Security*, pages 89–116. Springer, 2008. 47

[87] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999. 42

[88] FuSyS. Kstat. `http://www.s0ftpj.org/tools/kstat24_v1.1-2.tgz`. 86

[89] Debin Gao, Michael K. Reiter, and Dawn Song. Gray-box extraction of execution graphs for anomaly detection. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 318–329, New York, NY, USA, 2004. ACM. 36

[90] Debin Gao, Michael K. Reiter, and Dawn Xiaodong Song. On Gray-Box Program Tracking for Anomaly Detection. In *USENIX Security Symposium*, pages 103–118, 2004. 36

[91] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition based Security Tools. In *Proc. Network and Distributed Systems Security Symposium*, February 2003. 42

[92] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*, May 2007. 22, 47

[93] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA.*, pages 193–206. ACM, October 2003. 52

[94] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS*. The Internet Society, 2004. 42

[95] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 2003 Network and Distributed System Symposium*, 2003. 24, 43

[96] Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, Marcel Winandy, Rani Husseiki, and Christian Stüble. Flexible and secure enterprise rights management based on trusted virtual domains. In *STC '08: Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 71–80, New York, NY, USA, 2008. ACM. 55

[97] Pawel Gepner and Michal F. Kowalik. Multi-core processors: New way to achieve high system performance. In *PARELEC '06: International symposium on Parallel Computing in Electrical Engineering*, pages 9–13, Washington, DC, USA, 2006. IEEE Computer Society. 173

[98] Jonathon T. Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P. Miller. Environment-Sensitive Intrusion Detection. In *RAID*, pages 185–206, 2005. 37

[99] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Detecting Manipulated Remote Call Streams. In *Proceedings of the 11th USENIX Security Symposium*, pages 61–79, Berkeley, CA, USA, 2002. USENIX Association. 36

[100] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*. The Internet Society, 2004. 36

[101] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Automated Discovery of Mimicry Attacks. In Diego Zamboni and Christopher Krgel, editors, *RAID*, volume 4219 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2006. 39

[102] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th Usenix Security Symposium*, pages 1–13, San Jose, CA, USA, 1996. 41

[103] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, New York, NY, USA, 1973. ACM Press. 14

[104] R. P. Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(6):34–45, 1974. 14

[105] K. Goldman, R. Perez, and R. Sailer. Linking remote attestation to secure tunnel endpoints. *Proceedings of the first ACM workshop on Scalable trusted computing*, pages 21–24, 2006. 54

[106] Rajeev Gopalakrishna, Eugene H. Spafford, and Jan Vitek. Efficient Intrusion Detection using Automaton Inlining. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 18–31, Washington, DC, USA, 2005. IEEE Computer Society. 37

[107] J.L. Griffin, T. Jaeger, R. Perez, R. Sailer, L. van Doorn, and R. Caceres. Trusted Virtual Domains: Toward secure distributed services. *Proc. of 1st IEEE Workshop on Hot Topics in System Dependability (HotDep)*, 2005. 53

[108] Liang Gu, Xuhua Ding, Robert H. Deng, Yanzhen Zou, Bing Xie, Weizhong Shao, and Hong Mei. Model-Driven Remote Attestation: Attesting Remote System from Behavioral Aspect. *Young Computer Scientists, International Conference for*, 0:2347–2353, 2008. 51

[109] Shay Gueron and Jean-Pierre Seifert. On the impossibility of detecting virtual machine monitors. In Dimitris Gritzalis and Javier López, editors, *Emerging Challenges for Security, Privacy and Trust, 24th IFIP TC 11 International Information Security Conference, SEC 2009, Pafos, Cyprus, May 18-20, 2009. Proceedings*, volume 297 of *IFIP*, pages 143–151. Springer, 2009. 23

[110] Peter H. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM Journal of Research and Development*, 27(6):530–544, 1983. 14

[111] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association. 50

[112] Vivek Haldar and Michael Franz. Symmetric behavior-based trust: a new paradigm for internet computing. In *NSPW '04: Proceedings of the 2004 workshop on New security paradigms*, pages 79–84, New York, NY, USA, 2004. ACM. 50

[113] Ragib Hasan, Suvda Myagmar, Adam J. Lee, and William Yurcik. Toward a threat model for storage systems. In *StorageSS '05: Proceedings of the 2005 ACM workshop*

on Storage security and survivability, pages 94–102, New York, NY, USA, 2005. ACM Press. 56

[114] Brian Hay and Kara Nance. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82, 2008. 45

[115] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. *Lecture Notes in Computer Science*, 2648:235–240, 2003. xix

[116] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998. xviii, 35

[117] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 39(4):229–243, 2004. 61

[118] T.W. Hou, H.Y. Chen, and M.H. Tsai. Three control flow obfuscation methods for Java software. *IEE Proceedings-Software*, 153(2):80, 2006. 52

[119] V.M. Inc. VMware. `http://www.vmware.com/`. 16

[120] IOzone. Filesystem Benchmark. `http://www.iozone.org/`. 126, 167

[121] Daniel Jackson and Eugene J. Rollins. Chopping: A Generalization of Slicing. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994. 61

[122] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 19–28. ACM New York, NY, USA, 2006. 50

[123] Trent Jaeger, Kevin Butler, David H. King, Serge Hallyn, Joy Latten, and Xiaolan Zhang. Leveraging IPsec for mandatory access control across systems. In *Proceedings of the Second International Conference on Security and Privacy in Communication Networks*, August 2006. 53

[124] Trent Jaeger, Serge Hallyn, and Joy Latten. Leveraging IPsec for Mandatory Access Control of Linux Network Communications. Technical Report Technical Report RC23642, IBM T.J. Watson Research Center, April 2005. 53

[125] K. Jain and R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *Proc. Network and Distributed Systems Security Symposium*, 1999. 42

[126] B. Jansen, H.G.V. Ramasamy, and M. Schunter. Policy enforcement and compliance proofs for Xen virtual machines. *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 101–110, 2008. 55

[127] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138, New York, NY, USA, 2007. ACM. 44

[128] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 91–100, New York, NY, USA, 2008. ACM. 44

[129] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 91–104, New York, NY, USA, 2005. ACM Press. 43

[130] kad. Handling Interrupt Descriptor Table for fun and profit. *Phrack*, 11(59), July 2002. 102

[131] Nitin A. Kamble, Jun Nakajima, and Asit K. Mallick. Evolution in Kernel Debugging using Hardware Virtualization With Xen. In *Linux Symposium Proceedings*, pages 9–24, 2006. 43

[132] Gaurav S. Kc and Angelos D. Keromytis. e-NeXSh: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 286–302, Washington, DC, USA, 2005. IEEE Computer Society. 42

[133] Katarzyna Keahey, Karl Doering, and Ian Foster. From Sandbox to Playground: Dynamic Virtual Environments in the Grid. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 34–42, Washington, DC, USA, 2004. IEEE Computer Society. 53

[134] Vishal Kher and Yongdae Kim. Securing distributed storage: challenges, techniques, and systems. In *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 9–25, New York, NY, USA, 2005. ACM Press. 57

[135] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29, New York, NY, USA, 1994. ACM. xix

[136] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Trans. Comput. Syst.*, 23(1):51–76, 2005. 56, 163

[137] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing malware with virtual machines. *sp*, 0:314–327, 2006. 47

[138] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association. 42

[139] T. Klein. Scooby Doo - VMware Fingerprint Suite. `http://www.trapkit.de/research/vmm/scoopydoo/index.html`. 47

[140] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: a Specification-based approach. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 175, Washington, DC, USA, 1997. IEEE Computer Society. 35

[141] Kenichi Kourai and Shigeru Chiba. HyperSpector: virtual distributed monitoring environments for secure intrusion detection. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 197–207, New York, NY, USA, 2005. ACM. 43

[142] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In *Proceedings of the 8^{th} European Symposium on Research in Computer Security (ESORICS '03)*, LNCS, pages 326–343, Gjovik, Norway, October 2003. Springer-Verlag. 35

[143] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 11–11, Berkeley, CA, USA, 2005. USENIX Association. 39

[144] Dirk Kuhlmann, Rainer Landfermann, Hari V. Ramasamy, Matthias Schunter, Gianluca Ramunno, and Davide Vernizzi. An Open Trusted Computing Architecture Secure Virtual Machines Enabling User-Defined Policy Enforcement. Technical Report RZ3655, IBM Research, 2006. 50

[145] David Kyle and José Carlos Brustoloni. Uclinux: a Linux security module for trusted-computing-based usage controls enforcement. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing*, pages 63–70, New York, USA, 2007. ACM. 51

[146] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12, New York, NY, USA, 2009. ACM. 55

[147] Lap Chung Lam. *Program Transformation Techniques for Host-based Intrusion Prevention.* PhD thesis, Stony Brook University, 2005. 37

[148] Lap-Chung Lam and Tzi cker Chiueh. Automatic Extraction of Accurate Application-Specific Sandboxing Policy. In *RAID*, pages 1–20, 2004. xx, 37

[149] Lap Chung Lam, Wei Li, and Tzi cker Chiueh. Accurate and Automated System Call Policy-Based Intrusion Prevention. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 413–424, Washington, DC, USA, 2006. IEEE Computer Society. xx, 37

[150] M. Laureano, C. Maziero, and E. Jamhour. Protecting host-based intrusion detectors through virtual machines. *Comput. Networks*, 51(5):1275–1283, 2007. 43

[151] Wenke Lee and Salvatore J. Stolfo. Data mining approaches for intrusion detection. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, pages 79–94, Berkeley, CA, USA, 1998. USENIX Association. 35

[152] F. Leung, G. Neiger, D. Rodgers, A. Santoni, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3):167–178, August 2006. 22, 173

[153] Peng Li, Hyundo Park, Debin Gao, and Jianming Fu. Bridging the Gap between Data-Flow and Control-Flow Analysis for Anomaly Detection. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 392–401, Washington, DC, USA, 2008. IEEE Computer Society. 41

[154] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, New York, NY, USA, 2003. ACM. 52

[155] Lionel Litty and David Lie. Manitou: a layer-below approach to fighting malware. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 6–11, New York, NY, USA, 2006. ACM. 43

[156] Michael E. Locasto, Angelos Stavrou, Gabriela F. Cretu, Angelos D. Keromytis, and Salvatore J. Stolfo. Quantifying Application Behavior Space for Detection and Self-Healing. Technical Report CUCS-017-06, Department of Computer Science, Columbia University, April 2006. 38

[157] Hans Löhr, HariGovind V. Ramasamy, Ahmad-Reza Sadeghi, Stefan Schulz, Matthias Schunter, and Christian Stüble. Enhancing Grid Security Using Trusted Virtualization. In *ATC*, pages 372–384, 2007. 54

[158] P. A. Loscocco and S. D. Smalley. Meeting critical security objectives with security enhanced linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001. 154

[159] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association. 154

[160] Jonathan M. McCune, Trent Jaeger, Stefan Berger, Ramon Caceres, and Reiner Sailer. Shamon: A System for Distributed Mandatory Access Control. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 23–32, Washington, DC, USA, 2006. IEEE Computer Society. 53

[161] Jan Midtgaard and Thomas Jensen. A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In *SAS '08: Proceedings of the 15th international symposium on Static Analysis*, pages 347–362, Berlin, Heidelberg, 2008. Springer-Verlag. 38

[162] Jan Midtgaard and Thomas P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 287–298, New York, NY, USA, 2009. ACM. 38

[163] James Morris. Networking in NSA security-enhanced Linux. *Linux Journal*, 2005(129):3, 2005. 160

[164] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving Xen security through disaggregation. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 151–160, New York, NY, USA, 2008. ACM. 33

[165] Darren Mutz, William K. Robertson, Giovanni Vigna, and Richard A. Kemmerer. Exploiting Execution Context for the Detection of Anomalous System Calls. In *Recent Advances in Intrusion Detection, 10th International Symposium, RAID 2007, Gold Goast, Australia, September 5-7*, volume 4637 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2007. 38

[166] Kara Nance, Matt Bishop, and Brian Hay. Virtual Machine Introspection: Observation or Interference? *IEEE Security and Privacy*, 6(5):32–37, 2008. 45

[167] Netfilter.org. Netfilter/Iptables project. `www.netfilter.org/`. 87, 154

[168] NetworkWorld. EMC, Intel, VMware team to secure private clouds, 2010. `http://www.networkworld.com/news/2010/030110-emc-intel-vmware-clouds.html`. 51

[169] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 308–318, New York, NY, USA, 2008. ACM. 49

[170] NIST. Definition of Cloud Computing v15. `http://csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v15.doc`. 157

[171] Matunda Nyanchama and Slvia Osborn. Modeling mandatory access control in role-based security systems. In *In Database Security VIII: Status and Prospects. Chapman-Hall*, pages 31–40. Chapman & Hall, 1995. 151

[172] Chris Odhner. Security in NFS Storage Networks. Technical report, Network Appliance, February 2005. 159

[173] OpenVPN. An Open Source SSL VPN Solution. `http://openvpn.net/`. 118, 154

[174] R. Oppliger and R. Rytz. Does trusted computing remedy computer security problems? *Security & Privacy Magazine, IEEE*, 3(2):16–19, 2005. 50

[175] Andris Padegs. System/370 Extended Architecture: Design Considerations. *IBM Journal of Research and Development*, 27(3):198–205, 1983. 14

[176] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT), Montreal, Canada*. ACM, August 2009. 48

[177] Chetan Parampalli, R. Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *ASIACCS '08: Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 156–167, New York, NY, USA, 2008. ACM. 40

[178] Bryan Parno, Zongwei Zhou, and Adrian Perrig. Don't talk to zombies: Mitigating DDoS attacks via attestation. Technical Report CMU-CyLab-09-009, Carnegie Mellon University, jun 2009. 51

[179] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. *Security and Privacy, IEEE Symposium on*, 0:233–247, 2008. 44

[180] S. Pearson. Trusted Computing Platforms, the Next Security Solution. Technical Report HPL-2002-221, Trusted E-Services Laboratory & HP Laboratories Bristol, November 2002. 49, 110

[181] Adam G. Pennington, John D. Strunk, John Linwood Griffin, Craig A. N. Soules, Garth R. Goodson, and Gregory R. Ganger. Storage-based intrusion detection: watching storage activity for suspicious behavior. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 138–151, Berkeley, CA, USA, 2003. USENIX Association. 56

[182] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102, New York, NY, USA, 2009. ACM. 39

[183] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX Security Symposium*, pages 179–194, 2004. 48

[184] Nick L. Petroni, Jr., Timothy Fraser, AAron Walters, and William A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 289–304, Berkeley, CA, USA, 2006. USENIX Association. 51

[185] Nick L. Petroni, Jr. and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 103–115, New York, NY, USA, 2007. ACM. 37

[186] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A formal model for virtual machine introspection. In *VMSec '09: Proceedings of the 1st ACM workshop on Virtual machine security*, pages 1–10, New York, NY, USA, 2009. ACM. 46

[187] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974. 15

[188] J. Poritz, M. Schunter, E. Van Herreweghen, and M. Waidner. Property attestation: scalable and privacy-friendly security assessment of peer computers. Technical Report RZ3548, IBM Corporation, 2004. 50

[189] Danny Quist and Val Smith. Detecting the Presence of Virtual Machines Using the Local Data Table. Technical report, Offensive Computing, 2006. 47

[190] Nguyen Anh Quynh and Kuniyasu Suzaki. Xenprobes, a lightweight user-space probing framework for Xen virtual machine. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association. 44

[191] Nguyen Anh Quynh and Yoshiyasu Takefuji. Towards a tamper-resistant kernel rootkit detector. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 276–283, New York, NY, USA, 2007. ACM. 43

[192] Thomas Raffetseder, Christopher Krügel, and Engin Kirda. Detecting system emulators. In Juan A. Garay, Arjen K. Lenstra, Masahiro Mambo, and René Peralta, editors, *Information Security, 10th International Conference, ISC 2007, Valparaíso, Chile, October 9-12, 2007, Proceedings*, volume 4779 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2007. 48

[193] M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting. Authenticated system calls. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 358–367, June-1 July 2005. 36

[194] Hans P. Reiser and Rdiger Kapitza. VM-FIT: Supporting Intrusion Tolerance with Virtualisation Technology. In *Proceedings of the 1st Workshop on Recent Advances on Intrusion-Tolerant Systems (in conjunction with Eurosys 2007, Lisbon, Portugal, March 23, 2007)*, pages 18–22, 2007. 55

[195] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring. *Availability, Reliability and Security, International Conference on*, 0:74–81, 2009. 45

[196] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag. 45

[197] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *SSYM'00: Proceedings of the 9th*

conference on USENIX Security Symposium, pages 10–10, Berkeley, CA, USA, 2000. USENIX Association. 16, 17, 47

[198] J. Rutkowska. Subverting Vista Kernel For Fun And Profit. *BlackHat Briefings USA, August*, 2006. 48

[199] Ahmad-Reza Sadeghi and Christian Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *NSPW '04: Proceedings of the 2004 workshop on New security paradigms*, pages 67–77, New York, NY, USA, 2004. ACM. 49

[200] Hassen Saïdi. Guarded models for intrusion detection. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 85–94, New York, NY, USA, 2007. ACM. 38

[201] R. Sailer, X. Zhang, and T. Jaeger. Design and implementation of a TCG-based integrity measurement architecture. *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13 table of contents*, pages 16–16, 2004. 50

[202] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *Proceedings of the 2005 Annual Computer Security Applications Conference*, pages 276–285, December 2005. 53

[203] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 308–317, New York, NY, USA, 2004. ACM Press. 50

[204] Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert van Doorn, John Linwood Griffin, and Stefan Berger. sHype: A secure hypervisor approach to trusted virtualized systems. Technical Report RC23511, IBM Research, 2005. 53

[205] Gaspare Sala, Daniele Sgandurra, and Fabrizio Baiardi. Security and Integrity of a Distributed File Storage in a Virtual Environment. In *SISW '07: Proceedings of the Fourth International IEEE Security in Storage Workshop*, pages 58–69, Washington, DC, USA, 2007. IEEE Computer Society. xxiv

[206] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. 160

[207] Ravi Sandhu and Xinwen Zhang. Peer-to-peer access control architecture using trusted computing technology. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 147–158, New York, NY, USA, 2005. ACM. 50

[208] Luis Sarmenta. TPM/J: Java-based API for the Trusted Platform Module (TPM). `http://projects.csail.mit.edu/tc/tpmj/`. 118

[209] Dries Schellekens, Brecht Wyseur, and Bart Preneel. Remote attestation on legacy operating systems with trusted platform modules. *Sci. Comput. Program.*, 74(1-2):13–22, 2008. 51

[210] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 45, Washington, DC, USA, 2002. IEEE Computer Society. 52

[211] Love H. Seawright and Richard A. MacKinnon. VM/370 - A Study of Multiplicity and Usefulness. *IBM Systems Journal*, 18(1):4–17, 1979. 14

[212] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 144, Washington, DC, USA, 2001. IEEE Computer Society. 35

[213] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *SSYM'99: Proceedings of the 8th conference on USENIX Security Symposium*, pages 6–6, Berkeley, CA, USA, 1999. USENIX Association. 35

[214] R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 15–28, New York, NY, USA, 2003. ACM. 35

[215] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Externally verifiable code execution. *Commun. ACM*, 49(9):45–49, 2006. 50

[216] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 335–350, New York, NY, USA, 2007. ACM. 44

[217] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–16, New York, NY, USA, 2005. ACM. 50

[218] Monirul I. Sharif, Kapil Singh, Jonathon T. Giffin, and Wenke Lee. Understanding Precision in Host Based Intrusion Detection. In *RAID*, pages 21–41, 2007. 37

[219] Sriranjani Sitaraman and S. Venkatesan. Forensic Analysis of File System Intrusions Using Improved Backtracking. In *IWIA '05: Proceedings of the Third IEEE International Workshop on Information Assurance*, pages 154–163, Washington, DC, USA, 2005. IEEE Computer Society. 56

[220] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. Nai labs report, NAI Labs, Dec 2001. Revised May 2006. 160

[221] Alexey Smirnov and Tzi cker Chiueh. DIRA: Automatic Detection, Identification and Repair of Control-Hijacking Attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*. The Internet Society, 2005. xvi

[222] SourceForge.net. gtk-gnutella: The Graphical Unix Gnutella Client. `http://gtk-gnutella.sourceforge.net/`. 118, 122

[223] SourceForge.net. Trusted Boot. `http://sourceforge.net/projects/tboot`. 114

[224] Eugene H. Spafford. The internet worm program: an analysis. *SIGCOMM Comput. Commun. Rev.*, 19(1):17–57, 1989. xvi

[225] S. Sparks and J. Butler. Shadow Walker: Raising the bar for rootkit detection. *Black Hat Japan*, 2005. xix

[226] A. Srivastava, K. Singh, and J. Giffin. Secure Observation of Kernel Behavior. Technical Report GT-CS-08-01, Georgia Institute of Technology, 2008. 46

[227] Paul Stanton. Securing Data in Storage: A Review of Current Research. *CoRR*, 409034:2004, 2004. 56

[228] Stephanie Forrest and Steven A. Hofmeyr and Anil Somayaji and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996. xviii, 3, 35

[229] Mario Strasser and Heiko Stamer. A Software-Based Trusted Platform Module Emulator. In *Trust '08: Proceedings of the 1st international conference on Trusted Computing and Trust in Information Technologies*, pages 33–47, Berlin, Heidelberg, 2008. Springer-Verlag. 51

[230] Geoffrey Strongin. Trusted computing using amd "pacifica" and "presidio" secure virtual machine technology. *Inf. Secur. Tech. Rep.*, 10(2):120–132, 2005. 142

[231] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proc. of the 4th Symposium on Operating Design and Implementation (OSDI)*, 2000. 56

[232] Sufatrio and Roland H. C. Yap. Improving Host-Based IDS with Argument Abstraction to Prevent Mimicry Attacks. In *Recent Advances in Intrusion Detection, 8th International Symposium, RAID 2005, Seattle, WA, USA, September 7-9*, volume 3858 of *Lecture Notes in Computer Science*, pages 146–164. Springer, 2006. 40

[233] Francesco Tamberi, Dario Maggiari, Daniele Sgandurra, and Fabrizio Baiardi. Semantics-Driven Introspection in a Virtual Environment. In *IAS '08: Proceedings of the 2008 The Fourth International Conference on Information Assurance*

and Security, pages 299–302, Washington, DC, USA, 2008. IEEE Computer Society. xxiv

[234] G. Tandon and P. Chan. Learning Rules from System Call Arguments and Sequences for Anomaly Detection. *Workshop on Data Mining for Computer Security*, 2003. 35

[235] P. Traynor, M. Chien, S. Weaver, B. Hicks, and P. Mc Daniel. Non Invasive Methods for Host Certification. *ACM Trans. on Information and System Security, vol. 11, No. 3*, pages 1–23, 2008. 51

[236] Hsin-Yi Tsai, Yu-Lun Huang, and David Wagner. A graph approach to quantitative analysis of control-flow obfuscating transformations. *Trans. Info. For. Sec.*, 4(2):257–267, 2009. 52

[237] Timothy K. Tsai and Navjot Singh. Libsafe: Transparent System-wide Protection Against Buffer Overflow Attacks. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, page 541, Washington, DC, USA, 2002. IEEE Computer Society. xviii

[238] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Marting, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, May 2005. 151

[239] Alfonso Valdes and Diego Zamboni, editors. *Behavioral Distance for Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*. Springer, 2006. 39

[240] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009. 156

[241] G. Vigna. *Malware Detection*, chapter Static Disassembly and Code Analysis. Advances in Information Security. Springer, 2007. 130

[242] VMWare. VMsafe: A Security Technology for Virtualized Environments. `http://www.vmware.com/technology/security/vmsafe.html`. 26, 173

[243] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society. xx, 36, 39, 99

[244] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 255–264, New York, NY, USA, 2002. ACM. 5, 39

[245] David A. Wagner. *Static analysis and computer security: new techniques for software assurance*. PhD thesis, University of California at Berkeley, 2000. xx, 36

[246] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, University of Virginia, Charlottesville, VA, USA, 2000. 52

[247] Weichao Wang, Zhiwei Li, Rodney Owens, and Bharat Bhargava. Secure and efficient access to outsourced data. In *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 55–66, New York, NY, USA, 2009. ACM. 57

[248] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 545–554, New York, NY, USA, 2009. ACM. 46

[249] Christina Warrender, Stephanie Forrest, and Barak A. Pearlmutter. Detecting Intrusions using System Calls: Alternative Data Models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999. 35

[250] Jinpeng Wei, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, and Peng Ning. Managing security of virtual machine images in a cloud environment. In *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 91–96, New York, NY, USA, 2009. ACM. 58

[251] Lap-chung Lam Wei Li and Tzi cker Chiueh. Automatic Application-Specific Sandboxing for Win32/X86 Binaries. In *Proceedings of Program Analysis for Security and Safety Workshop (PASSWORD) co-located with ECOOP 2006, Nantes, France*, 2006. 37

[252] Paul D. Williams and Eugene H. Spafford. CuPIDS: An exploration of highly focused, co-processor-based information system protection. *Comput. Networks*, 51(5):1284–1298, 2007. 48

[253] S.J.Stolfo W.Lee and P.K.Chan. Learning patterns from UNIX processes execution traces for intrusion detection. In *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 50–56, 1997. AAAI Press. 35

[254] David Isaac Wolinsky, Abhishek Agrawal, P. Oscar Boykin, Justin Davis, Arijit Ganguly, Vladimir Paramygin, Peter Sheng, and Renato J. Figueiredo. On the Design of Virtual Machine Sandboxes for Distributed Computing in Wide Area Overlays of Virtual Workstations. In *First Workshop on Virtualization Technologies in Distributed Computing (VTDC)*, November 2006. 54

[255] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, Berkeley, CA, USA, 2002. USENIX Association. 160

[256] Zhuowei Li XiaoFeng Wang and Rui Wang. Leapfrog: Enhancing Information Protection in Commodity Applications with Dataflow Control. Technical Report TR670, Indiana University at Bloomington, 2005. 41

[257] Haizhi Xu, Wenliang Du, and Steve J. Chapin. Context Sensitive Anomaly Monitoring of Process Control Flow To Detect Mimicry Attacks and Impossible Paths. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID*, pages 21–38. Springer, 2004. 40

[258] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 291–304, New York, NY, USA, 2009. ACM. 41

[259] Aaram Yun, Chunhui Shi, and Yongdae Kim. On protecting integrity and confidentiality of cryptographic file system for outsourced storage. In *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 67–76, New York, NY, USA, 2009. ACM. 58

[260] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure coprocessor-based intrusion detection. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC*, pages 239–242, New York, NY, USA, 2002. ACM Press. 48

[261] Youhui Zhang, Yu Gu, Hongyi Wang, and Dongsheng Wang. Virtual-Machine-based Intrusion Detection on File-aware Block Level Storage. In *SBAC-PAD '06: Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'06)*, pages 185–192, Washington, DC, USA, 2006. IEEE Computer Society. 57

[262] Ming Zhao, Jian Zhang, and Renato J. Figueiredo. Distributed File System Virtualization Techniques Supporting On-Demand Virtual Machine Environments for Grid Computing. *Cluster Computing*, 9(1):45–56, 2006. 57

[263] Xin Zhao, Kevin Borders, and Atul Prakash. SVGrid: a secure virtual environment for untrusted grid applications. In *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing*, pages 1–6, New York, NY, USA, 2005. ACM Press. 53

[264] Xin Zhao, Kevin Borders, and Atul Prakash. Towards Protecting Sensitive Files in a Compromised System. In *SISW '05: Proceedings of the Third IEEE International Security in Storage Workshop (SISW'05)*, pages 21–28, Washington, DC, USA, 2005. IEEE Computer Society. 56